PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46 — A1

(11) International Publication Number: **WO 95/16236**

(43) International Publication Date: 15 June 1995 (15.06.95)

(21) International Application Number: PCT/US94/14067

(22) International Filing Date: 7 December 1994 (07.12.94)

(30) Priority Data:
08/165,265 10 December 1993 (10.12.93) US

(71) Applicant: CRAY RESEARCH, INC. [US/US]; 655A Lone Oak Drive, Eagan, MN 55121 (US).

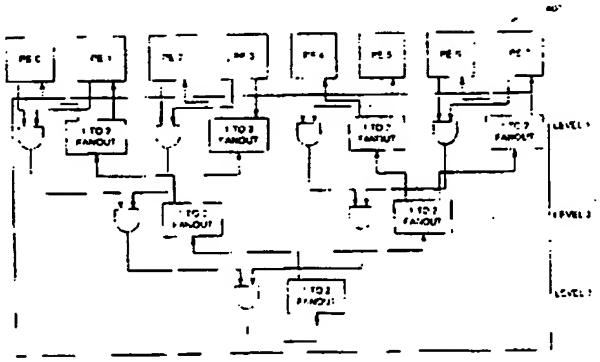
(72) Inventors: OBERLIN, Steven, M.; 20 Peterson Lane, Chippewa Falls, WI 54729 (US). FROMM, Eric, C.; 539 West Grand Avenue, Eau Claire, WI 54703 (US).

(74) Agent: RAASCH, Kevin, W.; Schwegman, Lundberg & Woessner, 3500 IDS Center, 80 South Eighth Street, Minneapolis, MN 55402 (US).

(81) Designated States: CA, JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: BARRIER SYNCHRONIZATION FOR DISTRIBUTED MEMORY MASSIVELY PARALLEL PROCESSING SYSTEMS



(57) Abstract

A barrier mechanism provides a low-latency method of synchronizing all or some of the processing elements (PEs) in a massively parallel processing system. The barrier mechanism is supported by one or more physical barrier synchronization circuits each receiving an input from every PE in the processing system. Each PE has an associated barrier synchronization register, in which each bit is used as an input to one of a plurality of logical barrier synchronisation circuits. The hardware supports both a conventional barrier function and an alternative eureka function. Each bit in the barrier synchronization registers can be programmed to perform as either barrier or eureka function, and all bits of the registers and each barrier synchronization circuits functions independently. Partitioning among PEs is accomplished by a barrier mask and interrupt register which enable certain of the bits in the barrier synchronization registers to a defined groups of PEs. Further partitioning is accomplished by providing bypass points in the physical barrier synchronization circuits to subdivide the physical barrier synchronization circuits into several types of PE barrier partitions of varying size and shape. The barrier mask and interrupt register and the bypass points are used in concert to accomplish flexible and scalable partitions corresponding to user-desired sizes and shapes with a latency several orders of magnitude faster than existing software implementations.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	GB	United Kingdom	MR	Mauritania
AU	Australia	GE	Georgia	MW	Malawi
BB	Barbados	GN	Guinea	NE	Niger
BE	Belgium	GR	Greece	NL	Netherlands
BF	Burkina Faso	HU	Hungary	NO	Norway
BG	Bulgaria	IE	Ireland	NZ	New Zealand
BJ	Benin	IT	Italy	PL	Poland
BR	Brazil	JP	Japan	PT	Portugal
BY	Belarus	KE	Kenya	RO	Romania
CA	Canada	KG	Kyrgystan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo			SE	Sweden
CH	Switzerland	KR	Republic of Korea	SI	Slovenia
CI	Côte d'Ivoire	KZ	Kazakhstan	SK	Slovakia
CM	Cameroon	LI	Liechtenstein	SN	Senegal
CN	China	LK	Sri Lanka	TD	Chad
CS	Czechoslovakia	LU	Luxembourg	TG	Togo
CZ	Czech Republic	LV	Latvia	TJ	Tajikistan
DE	Germany	MC	Monaco	TT	Trinidad and Tobago
DK	Denmark	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	US	United States of America
FI	Finland	ML	Mali	UZ	Uzbekistan
FR	France	MN	Mongolia	VN	Viet Nam
GA	Gabon				

BARRIER SYNCHRONIZATION FOR DISTRIBUTED MEMORY MASSIVELY PARALLEL PROCESSING SYSTEMS

Field of the Invention

The present invention relates generally to massively parallel processing systems, and more particularly to a barrier synchronization mechanism for facilitating synchronization between multiple processors in such a system.

Background of the Invention

Massively parallel processing (MPP) systems are computing systems comprised of hundreds or thousands of processing elements (PEs). While the power of a multiple instruction-multiple data (MIMD) computer system lies in its ability to execute independent threads of code simultaneously, the inherently asynchronous states of the PEs (with respect to each other) makes it difficult in such a system to enforce a deterministic order of events when necessary. Program sequences involving interaction between multiple PEs such as coordinated communication, sequential access to shared resources, controlled transitions between parallel regions, etc., may require synchronization of the PEs in order to assure proper execution.

An important synchronization capability in any programming model is the barrier. Barriers are points placed in the code beyond which no processor participating in a computation may proceed before all processors have arrived. Since processors wait at a barrier until alerted that all PEs have arrived, the latency of the barrier mechanism can be very important. The latency of a barrier mechanism is the propagation time between when the last processor arrives at a barrier, and when all processors have been notified that the barrier has been satisfied. During this period of time, all PEs may be idle waiting for the barrier to be satisfied. Hence barriers are a serialization point in a parallel code.

Barriers can be implemented entirely by software means, but software schemes are typically encumbered by long latencies and/or limited parallelism restricting how many processors can arrive at the barrier simultaneously without

artificial serialization. Because a barrier defines a serialization point in a program, it is important to keep the latency as low as possible.

Hardware support for barriers, while addressing the latency problems associated with barriers implemented by purely software means, can have other shortcomings that limit the utility of the mechanism in a production computing system. Production computing systems demand that the barrier resource (like all resources) be partitionable among multiple users while maintaining protective boundaries between users. In addition, the barrier resource must be rich enough to allow division between the operating system and the user executing within the same partition. Provision must also be made for fault tolerance to insure the robust nature of the barrier mechanism.

Hardware mechanisms may also suffer from an inability to operate synchronously. This inability may require that a PE, upon discovering that a barrier has been satisfied (all PEs have arrived at that point in the program), wait until all PEs have discovered that the barrier has been reached before it may proceed through the next section of program code. The ability to operate synchronously enables the barrier mechanism to be immediately reusable without fear of race conditions.

Hardware mechanisms may also require that a PE explicitly test a barrier flag to discover when the barrier condition has been satisfied. This can prevent a PE from accomplishing other useful work while the barrier remains unsatisfied, or force the programmer to include periodic tests of the barrier into the program in order to accomplish other useful work while a barrier is pending. This can limit the usefulness of a barrier mechanism when used as a means of terminating speculative parallel work (e.g., a database search) when the work has been completed (e.g. the searched-for item has been found).

Summary of the Invention

To overcome the above described shortcomings in the art and provide key system resources necessary for production computing, the present invention provides a hardware mechanism that facilitates barrier synchronization in a massively

parallel processing system. The present barrier mechanism provides a partitionable, low-latency, immediately reusable, robust mechanism which can be used to alert all PEs in a partition when all of the PEs in that partition have reached a designated point in the program. The mechanism permits explicit testing of barrier satisfaction, or can alternately interrupt the PEs when a barrier has been satisfied. The present invention also provides an alternate barrier mode, called a eureka, that satisfies a barrier when any one PE has reached a designated point in the program, providing the capability to terminate speculative parallel work. The present barrier mechanism provides multiple barrier resources to a partition to allow pipelining of barriers to hide the latency as well as offering raw latencies 2-3 orders of magnitude faster than software implementations. The barrier mechanism is also partitionable for multi-users. Barriers are used to bulk synchronize the processors in a partition between loops where producer/consumer hazards may exist, or control entry and exit between segments of a program.

Brief Description of the Drawings

The foregoing and other objects, features and advantages of the invention, as well as the presently preferred embodiments thereof, will become apparent upon reading and understanding the following detailed description and accompanying drawings in which:

FIGURE 1 shows a simplified block diagram of a representative MPP system with which the present barrier mechanism can be used;

FIGURE 2 shows a simplified block diagram of a processing element (PE), including a processor, local memory and associated shell circuitry;

FIGURE 3 shows the format of the barrier synchronization registers BSR0 and BSR1;

FIGURE 4 shows a simplified radix-2 barrier synchronization circuit;

FIGURE 5 shows a simplified radix-4 barrier synchronization circuit;

FIGURE 6 shows the format of the barrier synchronization function (BSFR) register;

FIGURE 7 shows the format of the barrier synchronization mask and interrupt (BSMI) register;

FIGURE 8 shows the bypass points in a simplified radix-2 barrier synchronization circuit;

FIGURE 9 shows the barrier synchronization circuit of FIGURE 8, with some of the bypass points redirected to the fanout block;

FIGURES 10A-E show how a 512 PE system can be partitioned at each level of a barrier synchronization circuit;

FIGURE 11 shows how the barrier synchronization mask and interrupt (BSMI) register is used in concert with the bypass points in the barrier synchronization circuits to achieve the desired shape and size of a particular partition;

FIGURE 12 shows the format of the barrier timing register BAR_TMG register;

FIGURE 13 shows a block diagram of a bypass point; and

FIGURE 14 shows the hardware state sequencing implementation for a single barrier bit.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In the following detailed description of the preferred embodiment, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration a specific embodiment in which the invention may be practiced. It is to be understood that the present detailed description is intended as exemplary only, and that other embodiments may be utilized and structural changes made without departing from the spirit and scope of the present invention.

The preferred MPP system, for which the present invention provides hardware support for barrier synchronization, is a MIMD massively parallel multiprocessor with a physically distributed, globally addressable memory. A representative MPP system 100 is shown in FIGURE 1. The MPP system 100

contains hundreds or thousands of processors, each accompanied by a local memory and associated support circuitry. Each processor, local memory and support circuity component is called a processing element (PE). The PE's in the MPP system 100 are linked via an interconnect network.

The preferred MPP system 100 has a physically distributed memory, wherein each processor has a favored, low latency, high bandwidth path to a local memory, and a longer latency lower bandwidth access to the memory banks associated with other processors over the interconnect network. In the preferred embodiment, the interconnect network is comprised of a 3-dimensional torus which when connected creates a 3-dimensional matrix of PEs. The torus design has several advantages, including speed of information transfers and the ability to avoid bad communication links. The interconnect network is also scalable in all three dimensions. The interconnect network is described in more detail in the copending and commonly assigned U.S. Patent Application Serial Number 07/983,979, entitled "Direction Order Routing in Multiprocessing Systems", to Gregory M. Thorsen, filed November 30, 1992, which is incorporated herein by reference.

FIGURE 2 shows a simplified block diagram of a PE 200. An individual PE includes a high-performance RISC (reduced instruction set computer) microprocessor 202. In the preferred MPP system, microprocessor 202 is the DECChip 21064-AA RISC microprocessor, available from Digital Equipment Corporation. Each PE is coupled to a local memory 204 that is a distributed portion of the globally-addressable system memory, and includes a shell of circuitry that implements synchronization and communication functions facilitating interactions between processors.

The shell circuitry includes an interconnection network router 206, used to connect multiple PEs in the three-dimensional toroidal "fabric". The interconnection network carries all data communicated between PEs and memories that are not local. An address centrifuge and block transfer engine 208 in the PE shell circuitry permits asynchronous (i.e., independent of the local processor) movement of data between the local memory 204 and remote memories associated

with other PEs, such as block transfers, with flexible addressing modes that permit a high degree of control over the distribution of data between the distributed portions of the system memory. The address centrifuge and block transfer engine are described in detail in the copending and commonly assigned U.S. Patent Application entitled "RECURSIVE ADDRESS CENTRIFUGE FOR DISTRIBUTED MEMORY MASSIVELY PARALLEL PROCESSING SYSTEMS", filed on even date herewith to Fromm, which application is incorporated herein by reference.

The shell circuitry also includes a data prefetch queue 210 which allows the processor 202 to directly initiate data movement between remote memories and the local processor in a way that can hide the access latency and permit multiple remote memory references to be outstanding.

Synchronization circuits in the shell permit synchronization at various different levels of program or data granularity in order to best match the synchronization method that is "natural" for a given parallelization technique. At the finest granularity, data-level synchronization is facilitated by an atomic swap mechanism that permits the locking of data on an element-by-element basis. A more coarse grain data-level synchronization primitive is provided by a messaging facility, which permits a PE to send a packet of data to another PE and cause an interrupt upon message arrival, providing for the management of message queues and low-level messaging protocol in hardware. Control-level synchronization at the program loop level is provided by a large set of globally accessible fetch-and-increment registers that can be used to dynamically distribute work (in the form of iterations of a loop, for instance) among processors at run time. The present invention provides yet another form of control-level synchronization, barrier synchronization, which is useful to control transitions between major program blocks (i.e., between loops performing actions on the same data sets).

Barrier Synchronization

The present invention provides hardware support for barrier synchronization which results in a low-latency method of synchronizing all or a

portion of the PEs in an MPP system. The barrier synchronization mechanism of the present invention may be used to perform two types of synchronization: barrier and eureka.

A barrier is a point in program code where, after reaching the barrier, a processor must wait until all other processors participating in the computation have also reached the barrier. After all of the processors reach the barrier, the processors continue issuing program code.

A programmer may place a barrier in a program between distributed, parallel loops performing operations on the same data. By doing this, the programmer ensures that all of the processors associated with a loop finish the loop (all writes to shared data complete) before any of the processors continue with other program instructions (access the new values of the shared data).

A eureka is a point in program code where a condition is established that only a single processor need satisfy, thereby causing all processors to proceed beyond the eureka point. To participate in a eureka event, all processors initialize the eureka barrier mechanism described herein and enable an interrupt, then proceed executing program code to solve for the eureka condition. As soon as any processor completes the computation, it triggers the eureka, thus causing an interrupt to all PEs. The interrupt indicates that the eureka has been satisfied and all PEs may continue beyond the eureka point.

Eureka synchronization has several uses, including database searches. Using eureka synchronization, a programmer can stop a database search as soon as any processor succeeds in finding the desired data rather than waiting for all of the processors to exhaust the search.

Logical Barrier Synchronization Circuits

The preferred barrier mechanism has 16 logical barrier synchronization circuits. Each PE in the MPP system has an input to each of the 16 logical barrier synchronization circuits. The multiple barrier synchronization circuits facilitate

partitioning between users and the operating system, as well as providing redundant resources for fault tolerance.

The inputs to the 16 logical barrier synchronization circuits are 16 bits which are contained in two special registers. Each PE contains two 8-bit registers called barrier synchronization register 0 (BSR0) 300 and barrier synchronization register 1 (BSR1) 301. FIGURE 3 shows the format of BSR0 300 and BSR1 301. Each of the 16 bits which comprise BSR0 300 and BSR1 301 is an input to one of the 16 logical barrier synchronization circuits. Thus, each PE has an input to each of the 16 logical barrier synchronization circuits.

Barrier synchronization register 0 (BSR0) 300 is an 8-bit, readable and writable, general access register. Preferably, BSR0 300 contains the eight least significant barrier bits for a PE. When read from, the value of BSR0 300 represents the value of bits 2' through 20 of BSR0 300. The remaining bits are not valid. Table 1 shows the bit format of BSR0 300 and describes each bit of the register.

Table 1

BSR0 300 Format

Bits	Name
2^0	Barrier Bit 2^0
2^1	Barrier Bit 2^1
2^2	Barrier Bit 2^2
2^3	Barrier Bit 2^3
2^4	Barrier Bit 2^4
2^5	Barrier Bit 2^5
2^6	Barrier Bit 2^5
2^7	Barrier Bit 2^7
2^{63}-2^8	These bits are not used.

Barrier synchronization register I (BSR1) 301 is an 8-bit, readable and writable, privileged access register. BSR1 301 contains the eight most significant barrier bits for a PE. When read from, the value of BSR1 301 represents the value of bits 2^{15} through 2^8 of BSR1 301 and 2^7 through 2^0 of BSR0 300. The remaining bits are not valid. Table 2 shows the bit format of BSR1 301 and describes each bit of the register.

Table 2

BSR1 301 Format

Bits	Name
2^7-2^0	These bits are not used; however, when BSR1 301 is read, these bits contain the value of bits 2^7 through 2^0 of BSR0 300.
2^8	Barrier Bit 2^8
2^9	Barrier Bit 2^9
2^{10}	Barrier Bit 2^{10}
2^{11}	Barrier Bit 2^{11}
2^{12}	Barrier Bit 2^{12}
2^{13}	Barrier Bit 2^{13}
2^{14}	Barrier Bit 2^{14}
2^{15}	Barrier Bit 2^{15}
2^{63}-2^{16}	These bits are not used.

All 16 of the logical barrier synchronization circuits function identically and operate independently. An example of the operation of the barrier synchronization circuits will now be given, using bit 2^2 of BSR0 300 when it is used for barrier synchronization and for eureka synchronization as an example for purposes of illustration.

It shall be understood that the remaining bits function in the same way as bit 2^2 in the following discussion. Each logical barrier synchronization circuit is implemented as an AND-tree and fanout-tree circuit. FIGURE 4 shows a barrier synchronization circuit 400 in a simplified MPP system using 2-input AND gates to form the AND-tree. For simplicity of illustration, the MPP system shown in FIGURE 4 contains only eight PEs. It shall be understood, however, that the present barrier mechanism and barrier synchronization circuits can be used with a system having any number of PEs.

Because the network in FIGURE 4 is accommodating 8 processors, PE0-PE7, and the AND-tree is implemented with 2-input AND gates, $\log_2 8=3$ levels of AND-tree are required to arrive and a final barrier bit representing the logical product of all the PEs' barrier bits.

As a starting condition, before barrier synchronization begins, bit 2^2 of BSR0 300 is reset to 0 in all of the PEs. When a processor satisfies the barrier condition, that processor sets bit 2^2 of its associated BSR0 300 register to 1. This action sends a 1 to one of the AND gates in the first layer of the AND-tree.

The first layer of the AND-tree shown in FIGURE 4 contains four AND gates. Each AND gate receives signals from two PEs. For example, one AND gate may receive signals from bit 2^2 of BSR0 300 in PE 0 and bit 2^2 of BSR0 300 in PE 1. When all of the processors reach the barrier point in the program code (satisfy the barrier) they have each set bit 2^2 of their associated BSR0 300 to 1, causing the output of each of the four AND gates in the first level of the AND-tree to switch to 1.

The second level of the AND-tree of FIGURE 4 contains two AND gates. Each AND gate receives signals from two of the AND gates in the first level of the AND tree. When the output of all of the AND gates in the first level of the AND-tree are 1, the output of both the AND gates in the second level of the AND-tree are 1.

The third level of the AND-tree contains the final AND gate. This AND gate receives signals from both AND gates in the second level of the AND-tree. When the output of both AND gates in the second level of the AND-tree are 1, the output of the final AND gate is 1. The output of the final AND gate sends an input to the fanout-tree circuit. The fanout tree circuit is used to report the result of the AND-tree back to PEs.

The first fanout block in the fanout tree receives a 1 from the single level 3 AND gate. After creating two copies of the 1, the first fanout block sends the 1's to the two fanout blocks in the second level of the fanout tree.

The two fanout blocks in the second level of the fanout tree each create two copies of the 1. The two fanout blocks in the second level of the fanout tree then sends the l's to four fanout blocks in the first level of the fanout tree.

The four fanout blocks in the first level of the fanout tree each create two copies of the 1. The fanout blocks in the first level of the fanout tree then send the l's to the eight PEs. This signals all of the PEs in the system that all of the processors have reached the barrier and the processor in the PE may continue with other program instructions. Because the fanout tree is dependent on the AND-tree, the fanout tree will report that all of the PEs have reached the barrier only when each PE in the system has set bit 2^2 to 1.

As will be described more fully below in connection with FIGURE 14, the barrier mechanism is designed to be immediately reusable. This means that as soon as a processor detects that the barrier bit has cleared (all processors have arrived at the barrier), it can immediately set the barrier bit again to announce its arrival at the next barrier if applicable. Doing so does not affect the notification of the prior barrier satisfaction to any other PE.

Eureka Synchronization

A logical barrier synchronization circuit 400 may also be used to perform eureka synchronization. One use of the eureka function is for the synchronization of a distributed, parallel data search. Once one of the PEs has located the targeted information, the successful PE can set a eureka bit to inform the other PEs involved that the search is completed.

To enable the eureka function, each PE contains a register called the barrier synchronization function register (BSFR). FIGURE 6 shows the format of the BSFR and how it functions in conjunction with the BSR0 300 and BSR1 301 registers. The barrier synchronization function register (BSFR) is preferably a 16-bit, write-only, system privileged register. The BSFR contains a 16-bit mask that indicates which bits of BSR0 300 and BSR1 301 are used for barrier synchronization and which bits are used for eureka synchronization. Bits 0-7 of the BSFR control

the function of BSR0 300 and bits 8-15 of the BSFR control BSR1 301. If a bit of the BSFR is set to 1, the corresponding bit of BSR0 300 or BSR1 301 is used for eureka synchronization. If a bit of the BSFR is set to 0, the corresponding bit of BSR0 300 or BSR1 301 is used for barrier synchronization. Table 3 shows the bit format of BSFR and describes each bit of the register.

Table 3

BSFR Format

Bits	Name
2^7-2^0	These bits indicate which bits of BSR0 300 are used for eureka synchronization. For example, as shown in Figure 6, when bit 2^2 of the BSFR is set to 1, bit 2^2 of BSR0 300 is used for eureka synchronization. When bit 2^2 of the BSFR is set to 0, bit 2^2 of BSR0 300 is used for barrier synchronization.
2^{15}-2^8	These bits indicate which bits of BSR1 301 are used for eureka synchronization. For example, as shown in FIGURE 6, when bit 2^{13} of the BSFR is set to 1, bit 2^{13} of BSR1 301 is used for eureka synchronization. When bit 2^{13} of the BSFR is set to 0, bit 2^{13} of BSR1 301 is used for barrier synchronization.
2^{63}-2^{16}	These bits are not used.

Because each of the 16 logical barrier synchronization circuits operate completely independently of each other, any of the bits in each of the two barrier synchronization registers BSR0 300 and BSR1 301 for a particular PE can be programmed to function in the conventional barrier mode, or in eureka mode. As will be described below in connection with FIGURE 14, in eureka mode the output of the AND-tree is read directly by the PEs with no intervening synchronization logic such as the latch used in conventional barrier synchronization.

Because in eureka mode the output of the fanout tree is read directly by the PEs, the barrier hardware is not synchronous as it is in conventional barrier

mode, nor is a eureka bit immediately reusable. Using a barrier bit in eureka mode requires the use of a second bit programmed to function in barrier mode in order to prevent race conditions. The conventional barrier is used to synchronize the entry and exit of the PEs into and out of a eureka barrier. This implementation means that only half of the 16 barrier bits can be used for eureka synchronization.

Reading BSR0 300 or BSR1 301 returns the current state of the AND-tree. This allows the AND-tree to be used as an OR tree by using negative logic. After all PEs initialize their barrier bit to a logical 1, the output of the AND-tree can be read as a one by all PEs. If any PE writes a zero to its barrier bit, the output of the AND-tree will read as a zero, performing the OR function.

A typical eureka sequence begins with all processors initializing the eureka bit to a logical 1 and setting the conventional barrier bit to a logical 1. When the barrier switches to a zero (meaning all processors have initialized their eureka bit and joined the barrier), the eureka, bit is "armed". The processors then begin the parallel data search or other activity that is to be terminated by the eureka. When a processor satisfies the termination conditions, it clears its eureka bit to alert the other processors and sets its conventional barrier bit to indicate it has observed the eureka event. As each of the other PEs detect that the eureka event has occurred (because the output of the eureka AND-tree drops to a logical 0), it sets its conventional barrier bit and waits for the barrier to complete. When all PEs have joined the final barrier, they may proceed to arm the next eureka.

Servicing A Barrier

In the preferred embodiment of the present invention, the processor monitors the output of the fanout circuitry using one of two mechanisms: periodically testing the barrier bit (such as with a continuous loop) or enabling a barrier interrupt.

Continuing to use bit 2^2 as an example, in the first mechanism, after the processor sets bit 2^2 of BSR0 300 to 1, the processor may enter a loop that continuously checks the value of bit 2^2 of BSR0 300. After receiving a 1 from the

fanout circuitry, the support circuitry in the PE resets bit 2^2 of BSR0 300 to 0. Because the processor is regularly checking the value of bit 2^2 of BSR0 300, the processor may continue executing program instructions as soon as it is detected that bit 2^2 of BSR0 300 is reset to 0.

In the barrier interrupt mechanism, after the processor satisfies the barrier and sets bit 2^2 of BSR0 300 to 1, the processor enables a barrier interrupt. The processor may then issue program instructions that are not associated with the barrier. After receiving a 1 from the fanout circuitry, the support circuitry in the PE resets bit 2 of BSR0 300 to 0 and sets the barrier interrupt to the processor. The barrier interrupt indicates to the processor that all of the processors have reached the barrier, and causes the processor to suspend the unrelated activity and return to executing instructions associated with the barrier. The advantage of the barrier interrupt over continuous polling is the ability to perform other useful work while the other processors are approaching the barrier.

In the preferred embodiment, the microprocessor enables the barrier hardware interrupt using a hardware interrupt enable register (HIER) in the microprocessor system control register. For more information on the HIER and the system control register, refer to the DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet, available from Digital Equipment Corporation, which is incorporated herein by reference. The DEC microprocessor has 6 inputs for external hardware interrupts. These inputs appear in the HIRR (Hardware Interrupt Request Register) in bit positions 5-7 and 10-12. One of these six inputs is designated as the Barrier Interrupt Request bit.

The HIRR inputs can be enabled or disabled by a mask bit located in the HIER (Hardware Interrupt Enable Register) internal to the microprocessor. For more information on the HIRR and HIER registers, refer to pages 3-26 through 3-29 in the Digital Equipment Corporation publication: EV-3 AND EV-4 SPECIFICATION Version 2.0 May 3, 1991, which is incorporated herein by reference.

Those skilled in the art are aware that all RISC microprocessors provide comparable facilities for allowing and controlling the direct sampling of external hardware interrupt signals, and that the present invention is not limited to use with the DEC RISC microprocessor described herein.

The interrupt input to the microprocessor is asserted whenever any of the barrier bits selected by the BSMI (barrier synchronization mask and interrupt, discussed below) make the transition from a logical 1 to a logical 0. The interrupt input to the microprocessor is cleared by writing a bit in the system control register. To assure that no satisfied barrier events are missed, the correct programming sequence would be to clear the interrupt, then read BSR0 300 and BSR1 301 to determine which bit(s) have toggled.

After the support circuity sets the barrier hardware interrupt, the processor must read the BSMI register (as described below) to determine if the interrupt was associated with BSR0 300 or BSR1 301. When the processor reads the value of the BSMI register, the support circuitry clears the interrupt associated with BSR0 300 and the interrupt associated with BSR1 301.

If a barrier interrupt occurs while the processor is reading the BSMI register, the interrupt still occurs and is not cleared. The processor must then read the value of the BSMI register again to determine if the interrupt was associated with BSR0 300 or BSR1 301 and to clear the interrupts.

Logical Partitions

Not all of the PEs in a multiprocessing system may need to be part of a barrier or eureka synchronization process. Also, it is often desirable to have several partitions of PEs operational on different tasks simultaneously. To facilitate this, each PE also contains a barrier synchronization mask and interrupt (BSMI) register which is used to enable or disable a logical barrier synchronization circuit for a PE. The BSMI register contains a mask that indicates which bits of BSR0 300 and BSR1 301 are enabled for the PE, thus defining which partition(s) a PE is a member of, and defining partitioning among the PEs.

The 16 independent barrier synchronization circuits visible to the user in the barrier registers can be assigned by the operating system to different groups or teams of processors. The BSMI allows the barrier bits in any given processor to be enabled or disabled. Alone, the BSMI permits the division of the barrier resource among up to 16 different partitions with arbitrary membership. Used in conjunction with the barrier network partitioning capability described herein below, the BSMI allows the flexible subdivision of the barrier resource among a very large number of partitions in a scalable fashion.

FIGURE 7 shows the format of the BSMI register. The BSMI is a 16-bit, readable and writable, system privileged register. When written to, the BSMI register controls which bits in BSR0 300 and BSR1 301 are enabled for use by the PE. Bits 7-0 of BSMI enable the bits in BSR0 300, while bits 15-8 enable the bits in BSR1 301. If a bit of the BSMI register is set to 1, the corresponding bit of BSR0 300 or BSR1 301 is enabled. If a bit of the BSMI register is set to 0, the corresponding bit of BSR0 300 or BSR1 301 is disabled.

The BSMI register has a different bit format when written to than it does when it is read from. Table 4 shows the bit format of the BSMI register when it is written to and describes each bit of the register.

A disabled BSR0 300 or BSR1 301 bit appears to the logical product network to be always satisfied (a logical "1"). This permits the barrier to function normally for other processors whose barrier bit is enabled. Reading a disabled barrier synchronization register bit returns a logical "0". Writing a disabled barrier synchronization register bit has no effect.

Table 4

BMSI Register Write Format

Bits	Name
2^7-2^0	These bits indicate which bits of BSR0 300 are enabled for use by the PE. For example, when bit 2^2 of the BSMI register is set to 1, as shown in FIGURE 7, bit 2^2 of BSR0 300 is enabled for use by the PE. When bit 2^2 of BSR0 300 is disabled and cannot be used by the PE.
2^{15}-2^8	These bits indicate which bits of BSR1 301 are enabled for use by the PE. For example, when bit 2^{13} of the BSMI is set to 1, as shown in FIGURE 7, bit 2^{13} of BSR1 301 is enabled for use by the PE. When bit 2^{13} of the BSMI is set to 0, bit 2^{13} of BSR1 301 is disabled and cannot be used by the PE.
2^{63}-2^{16}	These bits are not used.

Table 5 shows the bit format of the BSMI register when it is read from and describes each bit of the register. When read from, bits 2^{14} and 2^{15} of the BSMI register provide the current state of the barrier interrupts from BSR0 300 and BSR1 301, respectively. After being read the BSMI register is cleared.

Table 5

BSMI Register Read Format

Bits	Name
2^{13}-2^{0}	These bits are not valid.
2^{14}	This bit reflects the current state of the barrier interrupt associated with bits 2' through 20 of BSR0 300.
2^{15}	This bit reflects the current state of the barrier interrupt associated with bits 2 through 28 of BSR1 301.
2^{63}-2^{16}	These bits are not valid.

Software may use the BSMI register to allow any set number of PEs to use one of the logical barrier synchronization circuits, and thus set up a partition among the PEs. For example, software may set bit 2^2 of the BSMI register to 1 in only four of the PEs in an MPP system. In this case, only the four PEs with bit 2^2 of the BSMI register set to 1 may use the logical barrier synchronization circuit associated with bit 2^2 of BSR0 300. This creates a logical barrier partition among the four PEs.

The BSMI and BSFR registers can be used in concert to arrive at several different barrier synchronization mechanisms. Table 6 shows the effect of one bit from the BSMI register and the BSFR on the corresponding bit in BSR0 300 or BSR1 301.

Table 6

Barrier Mask and Barrier Function Register:

BMSI Bit Value	BSFR Bit Value	Writing to BSR0 300 or BSR1 301	Reading from BSR0 300 or BSR1 301	Synchronization Type
0	0	No effect	Returns a 1	Disabled
0	1	No effect	Returns a 1	Disabled
1	0	Writing 1 indicates the microprocessor has reached the barrier. Writing 0 has no effect.	Returns a 1 if waiting for barrier to complete. Returns a 0 when barrier is complete.	Barrier
1	1	Writing 1 indicates the microprocessor is ready for eureka synchronization. Writing 0 indicates the microprocessor has completed the eureka process.	Returns a 1 if waiting for eureka to occur. Returns a 0 when a eureka occurs.	Eureka

Physical Barrier Synchronization Circuits

Although each of the 16 bits in the BSR0 300 and BSR1 301 registers in each PE represent an input to one of 16 logical barrier synchronization circuits, the preferred embodiment of the present barrier synchronization mechanism does not contain 16 physical barrier synchronization circuits. Instead, in the preferred implementation, the system contains 4 physical barrier synchronization circuits. The 16 bits of BSR0 300 and BSR1 301 are time multiplexed into the four physical barrier synchronization circuits.

One exemplary physical barrier synchronization circuit is shown in FIGURE 5. The preferred barrier network is implemented using a radix-4 AND-tree and fanout tree. As shown, a 1024 PE system contains $\log_4 1024 = 5$ levels in a barrier synchronization circuit.

Table 7 shows the input to each of the four physical barrier synchronization circuits during each clock period (CP). Four CPs are required for the physical barrier synchronization circuits to receive the input from all 16 bits in BSR0 300 and BSR1 301.

The input registers to the logical barrier synchronization circuits (as shown and described below in connection with FIGURE 14) are completely parallel, so any number of PEs can set barrier bits in the same clock period without contention. All PEs are informed that a particular barrier has been reached simultaneously, although due to the time multiplexed nature of the inputs to the circuits different processors may be at different points in a spin-loop testing the barrier bit.

Table 7

Physical Barrier Synchronization Circuit Inputs

Circuit	First C	Second C	Third C	Fourth C
0	Bit 2^0 of BSR0	Bit 2^4 of BSR0	Bit 2^8 of BSR1	Bit 2^{12} of BSR1
1	Bit 2^1 of BSR0	Bit 2^5 of BSR0	Bit 2^9 of BSR1	Bit 2^{13} of BSR1
2	Bit 2^2 of BSR0	Bit 2^6 of BSR0	Bit 2^{10} of BSR1	Bit 2^{14} of BSR1
3	Bit 2^3 of BSR0	Bit 2^7 of BSR0	Bit 2^{11} of BSR1	Bit 2^{15} of BSR1

With the preferred radix-4 AND-tree implementation such as that shown in FIGURE 5, each level of the tree takes approximately one clock period in logic and another 1 to 1.5 clock periods in wire travel time to accomplish. This assumption allows the latency for a barrier between 1024 processors to be estimated and compared with known latencies for software barrier techniques.

Table 8 illustrates the number of barrier bits and the estimated number of clock periods at each level of a radix-4 tree such as the one shown in FIGURE 5 connecting 1024 PEs. The radix-four tree reduces the barrier bit count by a factor of four in each level.

Table 8.

Barrier bit logical product tree levels

Number of barrier bits	Radix-four logical product tree level
1024	Level one, clock period 0
256	Level two, clock period 2
64	Level three, clock period 4
16	Level four, clock period 6
4	Level five, clock period 8.5 (1.5 clock wire)
1	Level six, clock period 11 (1.5 clock wire)

From Table 8 it can be seen that eleven clock periods are consumed to perform the logical product of the barrier bits from 1024 PEs using a radix-four tree with two to two-and-a-half clock periods per level of delay. If it is further assumed that the necessary fan-out of the final logical product all 1024 PEs was performed using a series of one-to-four single-clock fan-outs, another eleven clock periods would elapse, bringing the total latency time of a barrier propagation to 22 clock periods.

From this, the projected performance impact of the radix-4 tree implementation is relatively straight-forward to predict. With a time multiplexed tree that is four bits wide, 4 cycles of the tree are required to update the values of all 16 barrier bits. This is in addition to the approximately 22 clocks of initial latency. The best case occurs when the last processor sets a barrier bit at the same time the 4-bit "slice" that bit is a member of is entered into the logical product network (refer again to Table 7), adding nothing to the tree latency of 22 clocks. The worst case occurs when the last processor just misses the best case, setting a bit one clock period after the slice that bit is a member of was entered into the network. When this happens, the slice must wait 4 clocks to be entered into the network and another 22 for the propagation delay of the network. The final result is a minimum time of

22 cycles, maximum of 26, and an average of around 24 cycles. The resulting delay is about 160 nanoseconds at a 6.6 nanosecond clock period, assuming that the barrier network is clocked at the same speed as the processor. This delay is several orders of magnitude faster than known software barrier implementations, which can approach 500 microseconds or higher. This significant advantage of the present barrier synchronization mechanism will be well appreciated by those of skill in the art.

Physical Partitions

Note in FIGURES 4 and 5 that the "shape" of the fanout tree closely matches the configuration of the AND-tree. In the preferred embodiment, the fan-in nodes performing the AND function are physically located on the same integrated circuits as the fanout blocks at each level of the tree. An AND gate and fanout block pair is called a bypass point. This advantage of location makes possible programmable subdivision, or physical partitioning, of each barrier synchronization network into independent subnetworks by rerouting the output of the AND gate in each bypass point.

FIGURE 8 shows bypass points 302, 304, 306, 308, 310, 312 and 314 in a simplified radix-2 barrier synchronization circuit. In the preferred embodiment, the output of the AND gate in each of the bypass points can be redirected so that the output of the AND gate connects to the fanout block in that bypass point. For example, FIGURE 9 shows the same barrier synchronization circuit as shown in FIGURE 8. However, the output of the AND gate in bypass points 306, 308 and 310 is redirected to the fanout block in those bypass point. This results in a partitioning of the physical barrier synchronization circuit into three smaller barrier synchronization circuits, designated by phantom lines 402, 404 and 406.

The first circuit designated by phantom line 402 contains a two-level AND-tree and two-level fanout tree. This circuit is created by redirecting the output of the AND gate in bypass point 316 to the fanout block in bypass point 0. This smaller circuit operates identically to the barrier synchronization circuit in FIGURE

8. However, this circuit receives an input from and sends an output to PEs 0 through 3 only.

The second and third circuits designated by phantom lines 404 and 406, respectively, each contain a one-level AND-tree and fanout tree. These circuits are created by redirecting the output of the AND gates in bypass points 306 and 308 to the fanout blocks in bypass points 306 and 308.

The bypass mechanism just described can be used to create several types of barrier partitions. FIGURE 10 shows the resulting partitions when the PEs, in a 1024-PE MPP system using a radix-4 AND-tree and fanout tree are partitioned at each level of a physical barrier synchronization circuit. Each bypass point in level 1, of a physical barrier synchronization circuit connects to four PEs for a total of 256 level 1 bypass points: If all 256 of these bypass points have the output of the AND gate redirected to the fanout block, the PEs in the system are divided into 256 4-PE partitions, as shown in FIGURE 10A.

If all 32 of the level 2 bypass points have the output of the AND gate redirected to the fanout block, the PEs in the system are divided into 64 16-PE partitions, as shown in FIGURE 10B.

If all 8 of the level 3 bypass points have the output of the AND gate redirected to the fanout block, the PEs in the system are divided into eight 128-PE partitions, as shown in FIGURE 10C.

If all 4 of the level 4 bypass points have the output of the AND gate redirected to the fanout block, the PEs in the system are divided into four 256-PE partitions, as shown in FIGURE 10D.

If both of the level 5 bypass points have the output of the AND gate redirected to the fanout block, the PEs in the system are partitioned as one 1024 PE partition, as shown in FIGURE 10E.

The partitions shown in FIGURE 10 are only a subset of those that can be achieved using the bypass mechanism of the present barrier invention. FIGURE 10 shown only those achieved by redirecting all of the bypass points at each level. However, any number of bypass points in any number of different levels

can be redirected to achieve a wide variety of bypass partitions. Thus, many different partitions of different sizes can be created simultaneously, in the manner such as that shown in FIGURE 9. The result is a very flexible and scalable barrier resource. It shall therefore be appreciated by those of skill in the art that different partitions within the barrier mechanism can have bypass points enabled at different levels, and that sub-partitions of the partitions can have their barrier circuits bypassed at different levels, etc., conferring a great degree of flexibility of final partition subdivision upon the barrier tree.

When a bypass point of a barrier synchronization circuit has the output of the AND gate redirected to the fanout block, the PEs in the barrier partition can still use all 16 bits in BSR0 300 and BSR1 301; however, because there are only four physical barrier synchronization circuits, creating a barrier partition affects the four of the 16 bits in BSR0 300 and BSR1 301 which are input to the barrier synchronization circuit. For example, if a level 4 bypass point in physical barrier synchronization circuit 2 has the output of the AND gate redirected to the fanout block, the barrier partition contains 256-PEs. Because of this barrier partition, bits 2^2, 2^6, 2^{10}, and 2^{14} of BSR0 300 and BSR1 301 (the bits that are time multiplexed into the same physical barrier synchronization circuit as described above with respect to Table 7) in each of the 256-PEs that are part of the barrier partition only affect the 256-PEs in the barrier partition. In the 256-PEs, these bits cannot be used for system-level barrier or eureka synchronization, but can be used only for partition-level barrier or eureka synchronization for the PEs in the partition. This is because the other 768 PEs in the 1024-PE system are not part of the partition created and thus barrier communication amongst all PEs is not available in that instance.

Note that, unlike barrier partitions created using the BSMI register described above, each of the PEs in a partition created when the barrier network is bypassed still is able to use all 16 inputs of the BSR0 300 and BSR1 301 registers, just as in the case of an undivided network. Each PE has all 16 barrier bits available to it because none are being masked to create the partition. This gives the characteristic of scalability to the barrier resource: each sub-network created through

bypassing the barrier circuits is an independently functioning copy of a barrier bit. Thus, the same bit in the barrier register can serve many separate partitions.

A further advantage of partitions created when the network is bypassed in a manner such as that shown in FIGURES 9 and 10 is that the latency of a barrier is reduced as levels of the fan-in/fan-out tree are avoided. The smaller the partition, the lower the latency.

Flexible Barrier Resource

As is well known by those skilled in the art, various processing tasks require partitions of particular sizes and shapes to achieve the most efficient performance. By redirecting the output of AND gates in different level bypass points, a physical barrier synchronization circuit may be divided into any combination of the previously described barrier partitions. However, the number and shape of the PEs in the partitions achieved with physical barrier partitioning alone may not exactly match the size and shape of partition desired.

The barrier network subdivisions are arranged so that they fall on power-of two boundaries coincident with the logical partitions created when the preferred 3-D torus interconnect network is subdivided. The 3-D torus interconnect network can be divided by two in any dimension, raising the probability of mismatches between the desired partitioning of the interconnect network and the partitioning of the barrier network achievable by means of the network bypass mechanism. In these cases, the BSMI register is used in concert with the physical barrier partitioning mechanism to accomplish the partitioning along the boundaries desired. The operating system can use the BSMI register to disable some of the bits in BSR0 300 and BSR1 301 for some of the PEs and enable these bits for the other PEs in the barrier partition to arrive at almost any desired group of PE partitions.

For example, it may be desirable to logically divide the 512-PE system and the associated 8×8×8 3-D torus network shown in the upper left of FIGURE 11 into eight equal-sized 64-PE partitions with dimensions 4×4×4 shown in the lower left of FIGURE 11. Because of the radix-4 implementation of the barrier

synchronization, the barrier synchronization circuits can only be divided by 4 using the bypass mechanism and the division is fixed in the dimensions that are affected. For example, the level 5 barrier synchronization circuit subdivision using the physical barrier partitioning will split the PEs into four 128-PE partitions each dimensioned 4×4×8 as shown at the top center of FIGURE 11. This is an insufficient level of subdivision to match the desired interconnect network partitioning already described. However, if the next level of physical barrier network partitioning is activated, it will split each partition by 4 again, into sixteen 32-PE partitions each dimensioned 4×2×4 as shown at the far right of FIGURE 11. This results in too much partitioning to match the desired group of eight 64-PE partitions.

To achieve the desired degree of partitioning, the barrier synchronization circuits are first physically partitioned using the level 5 bypass mechanism to achieve four 4×4×8 partitions. The final division of the by-8 dimension is accomplished using the appropriate bits of the BSMI register. To achieve this result, the appropriate bit of the BSMI register of half of the PEs are set to 0, while the remaining half have their corresponding BSMI bit set to 1. This results in dividing the partitions in half to arrive at the desired partitioning of that shown in the bottom center of FIGURE 11, which is a match for the desired partitioning shown in the lower left of FIGURE 11.

Bypass Circuit Implementation

FIGURE 12 shows a block diagram of a hardware implementation for a radix-4 bypass circuit. The four inputs from the previous level of bypass circuits, or from the BSR0 300 or BSR1 301 registers if the bypass circuit is a level 1 bypass circuit, are input to four-input AND gate 350. The AND function is performed and the result output to the A input to mux 354 and to the B input to mux 356, which together form a bypass switch 355.

The "B" input to mux 354 is hardwired to a logical "1". The A input to mux 356 is received from the next level of fan-out in the fan-out tree.

For normal barrier synchronization circuit operation, select signal "S" is set such that the A input to both mux 354 and mux 356 are selected. If the output of AND 350 is to be redirected to fan-out circuit 352, select signal "S" is set such that the B inputs to mux 357 and mux 356 are selected. Thus, the result of the AND 350 is selected by mux 356. In this manner, the physical barrier partitioning achieved by "short-circuiting" the bypass circuits is accomplished.

However, if the output of the AND is to be redirected to the fanout block, the select line of mux 356 will select such that the AND output transmitted to the fanout circuit 352 by mux 356.

Timing

Because a physical barrier synchronization circuit may be divided into partitions, and because the 16 barrier bits are actually implemented by time-multiplexing on 4 physical barrier circuits, the total time for a bit to propagate through the circuit is not constant. This timing inconsistency may cause a bit to be in the wrong position when BSR0 300 or BSR1 301 is read. For example, if the timing is not set up correctly for physical barrier synchronization circuit 0, a bit that was originally written to the 20 bit location in BSR0 300 may appear in the 2^4 bit location of BSR0 300 when BSR0 300 is read (refer again to Table 7).

Because the barrier synchronization circuits are time multiplexed, and each bit in a barrier register is updated only every four clock periods, it is necessary for the depth of the pipelined fan-in/fan-out tree to be a multiple of four clock periods deep regardless of how it is configured or bypassed. When a physical barrier synchronization circuit is divided into barrier partitions, the time needed for a bit to propagate through each physical barrier synchronization circuit is not consistent. Since bypassing a level can remove a non-multiple-of-four number of clock periods of circuitry, it is necessary to have some programmable delays to adjust the apparent depth of the barrier network pipeline to a multiple of four after a bypass is implemented. These programmable delays, capable of adding from 0 to 3 clocks of delay, are located at each PE and skew the signal arriving from the barrier network

for each of the 4 physical circuits. The delays are programmed by writing an 8-bit memory-mapped register called the Barrier Timing register (BAR_TMG). FIGURE 13 shows the bit format of the BAR_TMG register. This register is organized as four groups of two bits, each of which represents a delay from 0 to 3 clocks to be added to one of the four physical barrier synchronization circuit outputs to correct the pipeline depth.

The BAR_TMG register is an 8-bit, write-only, system privileged register. The BAR_TMG register controls the timing of each physical barrier synchronization circuit. Table 9 shows the bit format of the BAR_TMG register.

Table 9

BAR_TMG Bit Format

Bits	Name
2^1-2^0	These bits control the timing for physical barrier synchronization circuit 0.
2^3-2^2	These bits control the timing for physical barrier synchronization circuit 1.
2^5-2^4	These bits control the timing for physical barrier synchronization circuit 2.
2^7-2^6	These bits control the timing for physical barrier synchronization circuit 3.
2^{63}-2^8	These bits are not used.

As an example, the following procedure sets the BAR_TMG timing value for physical barrier synchronization circuit 0 in the PEs of a barrier partition. Before performing the procedure, Barrier hardware interrupt to the processors should be disabled.

1. Write a value of 1111_{16} to the BSMI register in all of the PEs to enable barrier bits 2^{12}, 2^{8}, 2^{4}, and 2^{0}.

2. Write a value of 1111_{16} to the BSFR in all of the PEs to set barrier bits 2^{12}, 2^{8}, 2^{4}, and 2^{0} to eureka mode.

3. Write a value of 1111_{16} to BSR0 300 and BSR1 301 in all of the PEs to start the eureka processes.

4. In one of the PEs write a value of 1110_{16} to BSR0 300 to indicate that bit 20 has completed a eureka.

5. In each of the PEs, read the value of BSR1 301 which contains the value of all 16 barrier bits and apply a software mask so that the only bits read are 2^{12}, 2^{8}, 2^{4}, and 2^{0} and the remaining bits are set to 0. The value read from BSR1 301 may be 1110_{16}, 1101_{16}, 1011_{16}, or 0111_{16}. If the value is not 1110_{16}, increment bits 2^{1} through 2^{0} of the BAR_TMG register by 1 and read the value of BSR1 301 again.

6. If the value read from BSR1 301 is now 1110_{16}, the timing is set up correctly. If the value is not 1110_{16}, increment bits 2^{1} through 2^{0} of the BAR_TMG register by 1 and read the value of BSR1 301 again. This process must be repeated until the value read from BSR1 301 is 1110_{16}, but should not need to be performed more than three times. This is because if it is correct, the value need not be incremented at all, and if it is not correct, incrementing by three will cause the BSR1 301 to run through all four possible values.

This procedure may be used to set the timing for any of the physical barrier synchronization circuits in a barrier partition. Table 10 lists the barrier bits affected and the write pattern for each physical barrier synchronization circuit.

Table 10

Timing Procedure Values

0	2^{12}, 2^{8}, 2^{4}, and 2^{0}	1110_{16}
1	2^{13}, 2^{9}, 2^{5}, and 2^{1}	2220_{16}
2	2^{14}, 2^{10}, 2^{6}, and 2^{2}	4440_{16}
3	2^{15}, 2^{11}, 2^{7}, and 2^{3}	8880_{16}

Barrier Synchronization Register Implementation

A single bit of the barrier registers BSR0 300 or BSR1 301, BMSI and BSFR associated with a PE is illustrated in FIGURE 14. As stated above, all 16 bits in the preferred barrier register function identically. For simplicity of illustration, only the circuitry associated with one bit is shown. Also, note that the barrier network is not shown, nor is the circuitry used to interface the barrier registers to the microprocessor.

In FIGURE 14, multiplexors are market "MUX" and have a select input "S" and a pair of data inputs "1" and "0" that are gated to the output depending on the state of the "S" input. Latches have a data input "D" and a clocked output "Q" and may have an enable input "E" that controls when new data is to be captured. If a latch does not have an "E" input, then new data is captured every clock.

The three principle barrier registers are identifiable as the mask latch 802 (for BSMI), the function latch 804, and the barrier register itself, latch 808 (for BSR0 300 or BSR1 301) . Data from the microprocessor can be entered into any of these latches via the processor writing to special memory-mapped addresses. Control hardware decodes the address to generate one of the control signals LD_MSK, LD_FCN, or LD_BAR. LD_MSK enables data to be entered the mask latch 802, LD_FCN enables data to be entered the function latch 804, and LD_BAR controls

the input multiplexor 806 that gates data into barrier register latch 808 every clock period. Data in the mask 802 and function 804 latches is held in the absence of the enable signals, while the barrier latch hold path is through multiplexors 810 and 806.

If mask latch 802 has been set to a 0 (disabling the barrier bit), data is inhibited from entering the barrier register latch 808 by forcing a constant 1 via OR gate 805. This forces a constant 1 into the barrier network as well, allowing other selected bits at other PEs to function normally. A 0 in the mask latch 802 also forces a 1 into the receive latch 812 through OR gate 811. This disables the local processor from "eavesdropping" on the barrier activities of other PEs while deselected and prevents spurious barrier interrupts from deselected barrier bits.

Function latch 804 controls multiplexors 814 and 810. If the function latch 804 is set to 0 (barrier mode), mux 814 delivers the current state of the barrier register while mux 810 causes the barrier register latch to remain set if it is currently set as long as the incoming barrier state is still a zero (i.e., the barrier has not been satisfied). When the incoming barrier state switches to a 1, indicating that all PEs have set their barrier bits, then the barrier register hold path is broken by AND gate 816 and barrier register 808 reverts to a 0 until it is set again by the processor. Thus, the barrier mode is a synchronous and immediately reusable mode.

If function latch 804 is set to a 1 (eureka mode), mux 814 continuously gates the incoming state of the barrier tree to the processor through 811 and 812 and, while mux 810 causes the barrier register latch to remain at whatever state, 0 or 1, that the processor sets it to. Thus, the eureka mode is asynchronous: any changes to the barrier registers flow through the barrier tree and are directly sampled by the processors, allowing the AND-tree of the barrier synchronization circuit to be used as an OR-tree for eureka barriers.

Although specific embodiments have been illustrated and described herein for purposes description of the preferred embodiment, it will be appreciated by those of ordinary skill in the art that a wide variety of alternate and/or equivalent implementations calculated to achieve the same purposes may be substituted for the specific embodiment shown and described without departing from the scope of the

present invention. Those of skill in the electrical and computer arts will readily appreciate that the present invention may be implemented in a very wide variety of embodiments. This application is intended to cover any adaptations or variations of the preferred embodiment discussed herein. Therefore, it is manifestly intended that this invention be limited only by the claims and the equivalents thereof.

What is claimed is:

1. A computer system (100) comprising:

a plurality of processing elements (PEs) (200) including a first processing element (PE) (200), wherein said first PE (200) comprises:

a barrier synchronization register one (BSR1) (301), said BSR1 (301) including a first BSR bit (340) and a second BSR bit (342), wherein said first BSR bit (340) and said second BSR bit (342) are set when said first PE (200) reaches a first barrier point and a second barrier point, respectively;

a first BSR output (401); and

a coupler which couples a value representative of said first BSR bit (340) to said first BSR output (401); and

a barrier synchronization circuit (BSC) (400) having a first physical barrier synchronization circuit (PBSC) (402), wherein said PBSC (402) comprises a plurality of bypass points including a first bypass point (302) and a second bypass point (310), wherein each of said bypass points comprise:

a fanin gate (330) having a plurality of fanin inputs and a fanin output, wherein said fanin output generates an indication of whether said fanin inputs are all set;

a fanout circuit (331) having a fanout input and a fanout output; and

a bypass switch (355) for switchably coupling said fanin output to said fanout input;

wherein one of the fanin inputs of said first bypass point (302) is coupled to said first BSR output (401), wherein one of the fanin inputs of said second bypass point (310) is coupled to the fanin output of said first bypass point (302), wherein the fanout input of said second bypass point (310) is coupled to the fanin output of said second bypass point (310), wherein the fanout input of said first bypass point

(302) is coupled to the fanout output of said second bypass point (310), and wherein the fanout output of said first bypass point (302) is coupled to said first PE (200).

2. The computer system according to claim 2, wherein

said coupler comprises a time multiplexor (TM), wherein said TM time-multiplexes said first BSR bit (340) and said second BSR bit (342) on said first BSR output (401).

3. The computer system according to claim 2, wherein said BSC 400 further comprises a second PBSC (402), wherein said second PBSC (402) comprises:

a third bypass point (302) and a fourth bypass point (310), wherein each of said bypass points comprise:

a fanin gate (330) having a plurality of fanin inputs and a fanin output which generates an indication of whether said fanin inputs are all set;

a fanout circuit (331) having a fanout input and a fanout output; and

a bypass switch (355) for switchably coupling said fanin output to said fanout input;

wherein said first PE (200) further comprises a second BSR output (401);

wherein said coupler further couples a value representative of said second BSR bit (342) to said second BSR output (401); and

wherein one of the fanin inputs of said third bypass point (302) is coupled to said second BSR output (401), wherein one of the fanin inputs of said fourth bypass point (310) is coupled to the fanin output of said third bypass point (302), wherein the fanout input of said fourth bypass point (310) is coupled to the fanin output of said fourth bypass point (310), wherein the fanout input of said third bypass point (302) is coupled to the fanout output of said fourth bypass point (310), and wherein the fanout output of said third bypass point (302) is coupled to said first PE (200).

4. The computer system according to claim 2, wherein said BSC 400 further comprises a second PBSC (402), wherein said second PBSC (402) comprises:

a third bypass point (302) and a fourth bypass point (310), wherein each of said bypass points comprise:

a fanin gate (330) having a plurality of fanin inputs and a fanin output which generates an indication of whether said fanin inputs are all set;

a fanout circuit (331) having a fanout input and a fanout output; and

a bypass switch (355) for switchably coupling said fanin output to said fanout input;

wherein said BSR1 (301) further includes a third BSR bit (341) and a fourth BSR bit (343) which are set when said first PE (200) reaches a third and a fourth barrier point, respectively;

wherein said first PE (200) further comprises a second BSR output (401);

wherein said coupler comprises a time multiplexor (TM), wherein said TM time-multiplexes said first BSR bit (340) and said second BSR bit (342) on said first BSR output (401), and wherein said TM time-multiplexes said third BSR bit (341) and said fourth BSR bit (343) on said second BSR output (401); and

wherein one of the fanin inputs of said third bypass point (302) is coupled to said second BSR output (401), wherein one of the fanin inputs of said fourth bypass point (310) is coupled to the fanin output of said third bypass point (302), wherein the fanout input of said fourth bypass point (310) is coupled to the fanin output of said fourth bypass point (310), wherein the fanout input of said third bypass point (302) is coupled to the fanout output of said fourth bypass point (310), wherein the fanout output of said third bypass point (302) is coupled to said first PE (200).

5. A computer system according to claim 2, wherein the fanout output of said first bypass point (302) generates an interrupt to said first PE (200).

6. A computer system according to claim 2, wherein the fanout output of said first bypass point (302) controls a value which is polled by said first PE (200).

7. A method for barrier synchronization on a computer system, said computer system having a plurality of processing elements (PEs) (200) including a first PE (200), and a physical barrier synchronization circuit (402) including a plurality of bypass points (302, 310) each having a fanin gate (330) and a fanout circuit (331), wherein said bypass points are arranged in a tree having a plurality of levels, each of said plurality of PEs (200) including a barrier synchronization register one (BSR1) (301) and a first BSR output (401), each said BSR1 (301) having a first BSR bit (340) set when its associated PE (200) reaches a first barrier point, the method comprising the steps of:

outputting a value representative of the first BSR bit (340) on the first BSR output (401) of each of said PEs 200;

fanning-in a value from the first BSR outputs (401) from a first plurality of said PEs (200) to generate a first barrier-completion signal (403);

selectively bypassing said first barrier completion signal (403) at one of the plurality of levels from the fanin gate (330) to the fanout circuit (331) in order to partition the tree; and

fanning-out said first barrier completion signal (403) to each of said first plurality of PEs (200).

8. A method according to claim 8, wherein each said BSR1 (301) further includes a second BSR bit (342) set when its associated PE (200) reaches a second barrier point, wherein:

the step of outputting comprises the step of time multiplexing said first BSR bit (340) and said second BSR bit (342).

9. A method according to claim 8, wherein each said BSR1 (301) further includes a second (342), third (341), and fourth BSR bit (343) set when its associated

PE (200) reaches a second, third and fourth barrier point, respectively, and a second BSR output (401), wherein the step of outputting comprises the steps of:

time multiplexing said first BSR bit (340) and said second BSR bit (342) to said first BSR output (401); and

time multiplexing said third BSR bit (341) and said fourth BSR bit (343) to said second BSR output (401).

10. A system for processor barrier synchronization of a plurality of processing elements (PEs) (200) in a distributed processing computer (100), comprising:

a barrier detection program means operative in each PE (200) to determine when each individual PE (200) has reached a processor barrier;

a plurality of boolean barrier synchronization register (BSR) outputs (401) from each PE (200), each output (401) controlled by the barrier detection program means, indicating when a processor barrier was reached by that PE (200);

a plurality of physical barrier synchronization circuits (402) for detecting which PEs (200) have reached a particular processor barrier, comprising:

a plurality of boolean logical AND gates (330) connected to corresponding BSR outputs (401) of the plurality of PEs (200) to indicate when all of the PEs (200) reach a processor barrier;

an ALLDONE signal output (403) from the plurality of AND gates for providing an output indicating that all of the PEs (200) have reached a particular processor barrier; and

a plurality of logical fanout devices (331), with inputs connected to the ALLDONE signal line and outputs connected to the multiplicity of PEs (200) to indicate to every PE (200) in the system that the particular processor barrier was reached; and

a plurality of barrier synchronization register bits (340, 341) for storing the status of each barrier synchronization process.

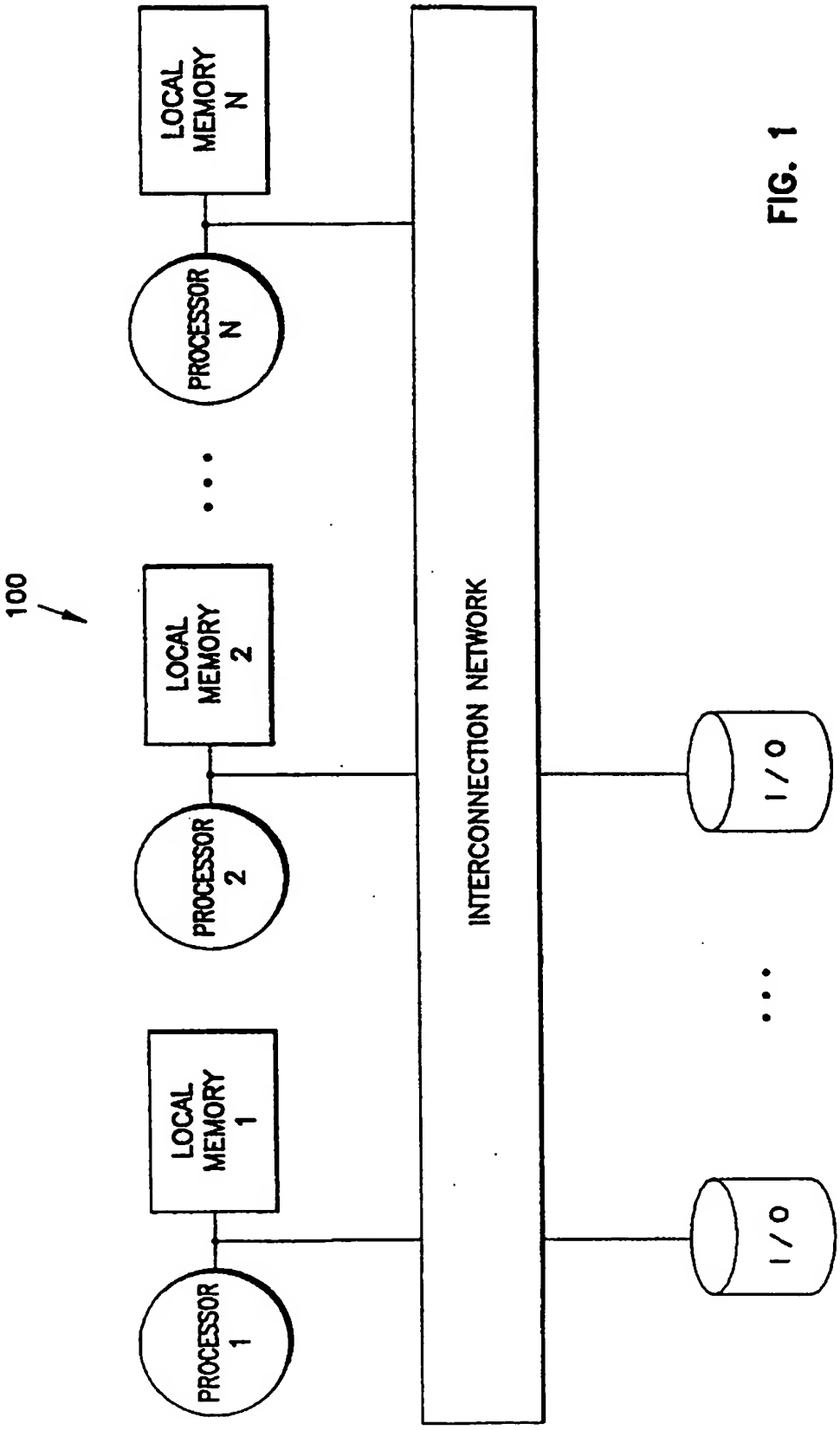

100
LOCAL MEMORY N
LOCAL MEMORY 2
LOCAL MEMORY 1
PROCESSOR N
PROCESSOR 2
PROCESSOR 1
INTERCONNECTION NETWORK
I / O
I / O

FIG. 1

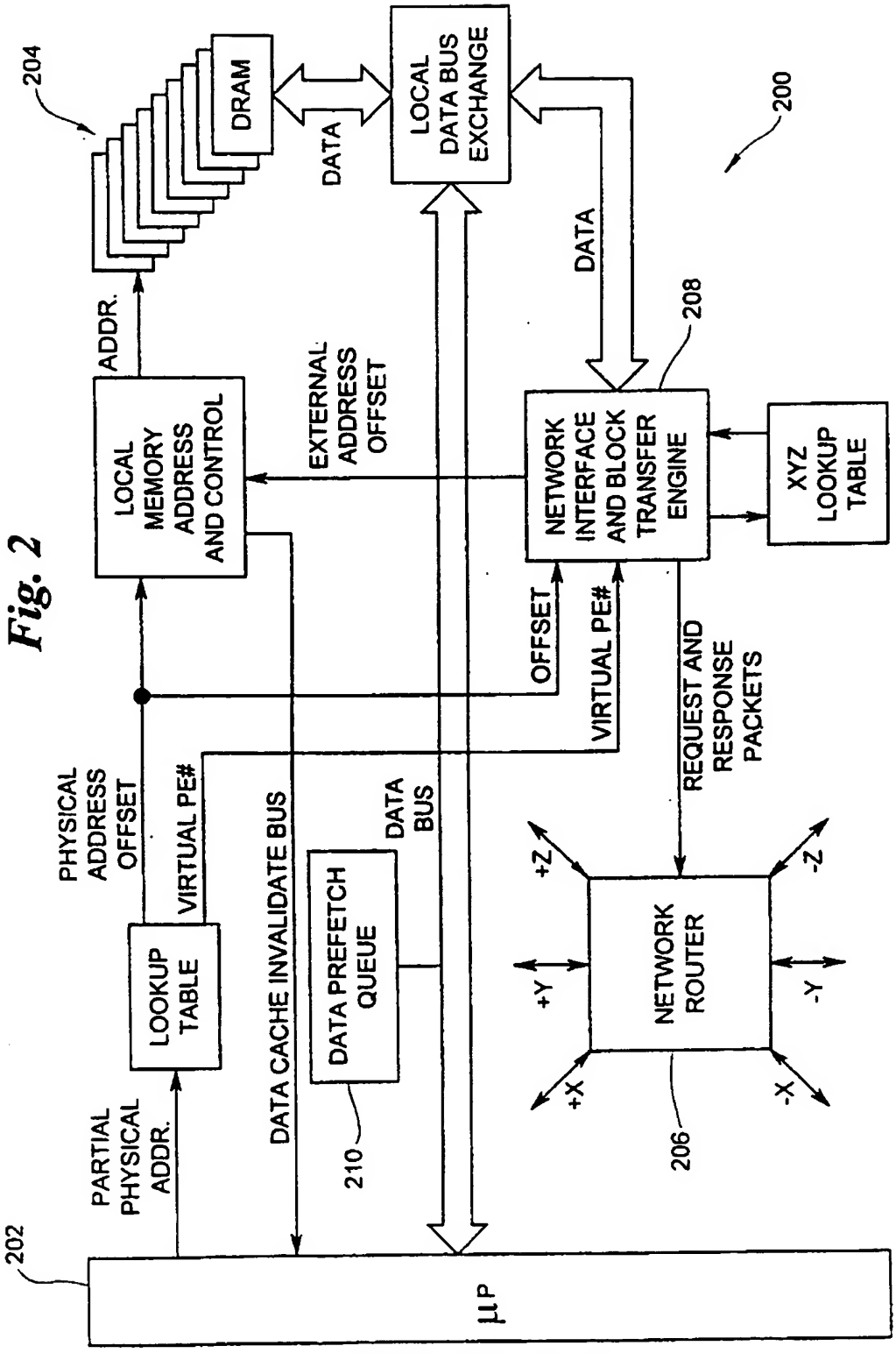

204
DRAM
DATA
LOCAL DATA BUS EXCHANGE
200
ADDR.
DATA
208
LOCAL MEMORY ADDRESS AND CONTROL
EXTERNAL ADDRESS OFFSET
NETWORK INTERFACE AND BLOCK TRANSFER ENGINE
XYZ LOOKUP TABLE
OFFSET
VIRTUAL PE#
REQUEST AND RESPONSE PACKETS
PHYSICAL ADDRESS OFFSET
VIRTUAL PE#
DATA CACHE INVALIDATE BUS
DATA BUS
+Z
-Z
+Y
-Y
+X
-X
NETWORK ROUTER
LOOKUP TABLE
DATA PREFETCH QUEUE
PARTIAL PHYSICAL ADDR.
210
206
202
μP

Fig. 2

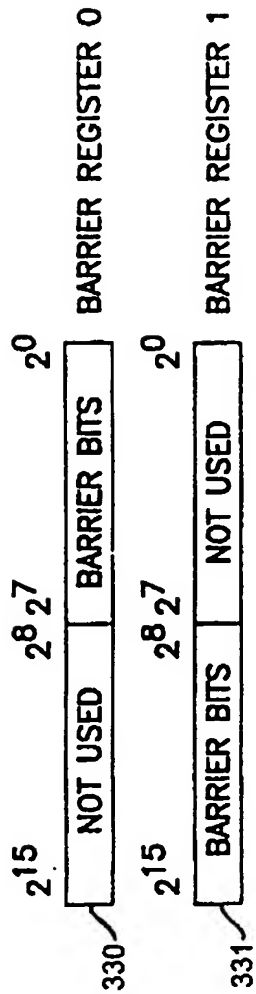

BARRIER REGISTER 0
BARRIER REGISTER 1
2^{15}
2^8 2^7
2^0
NOT USED
BARRIER BITS
BARRIER BITS
NOT USED
330
331

FIG. 3

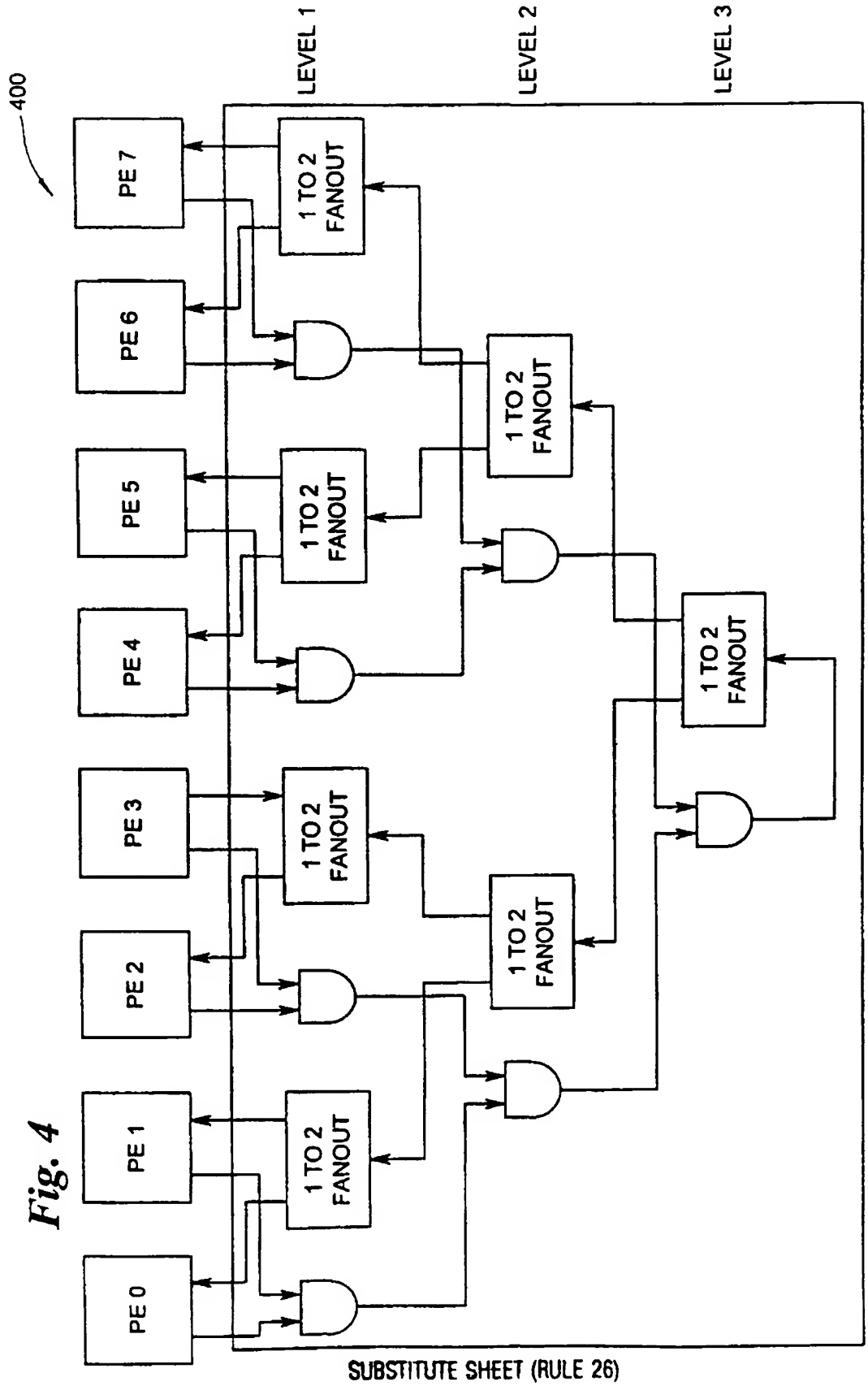
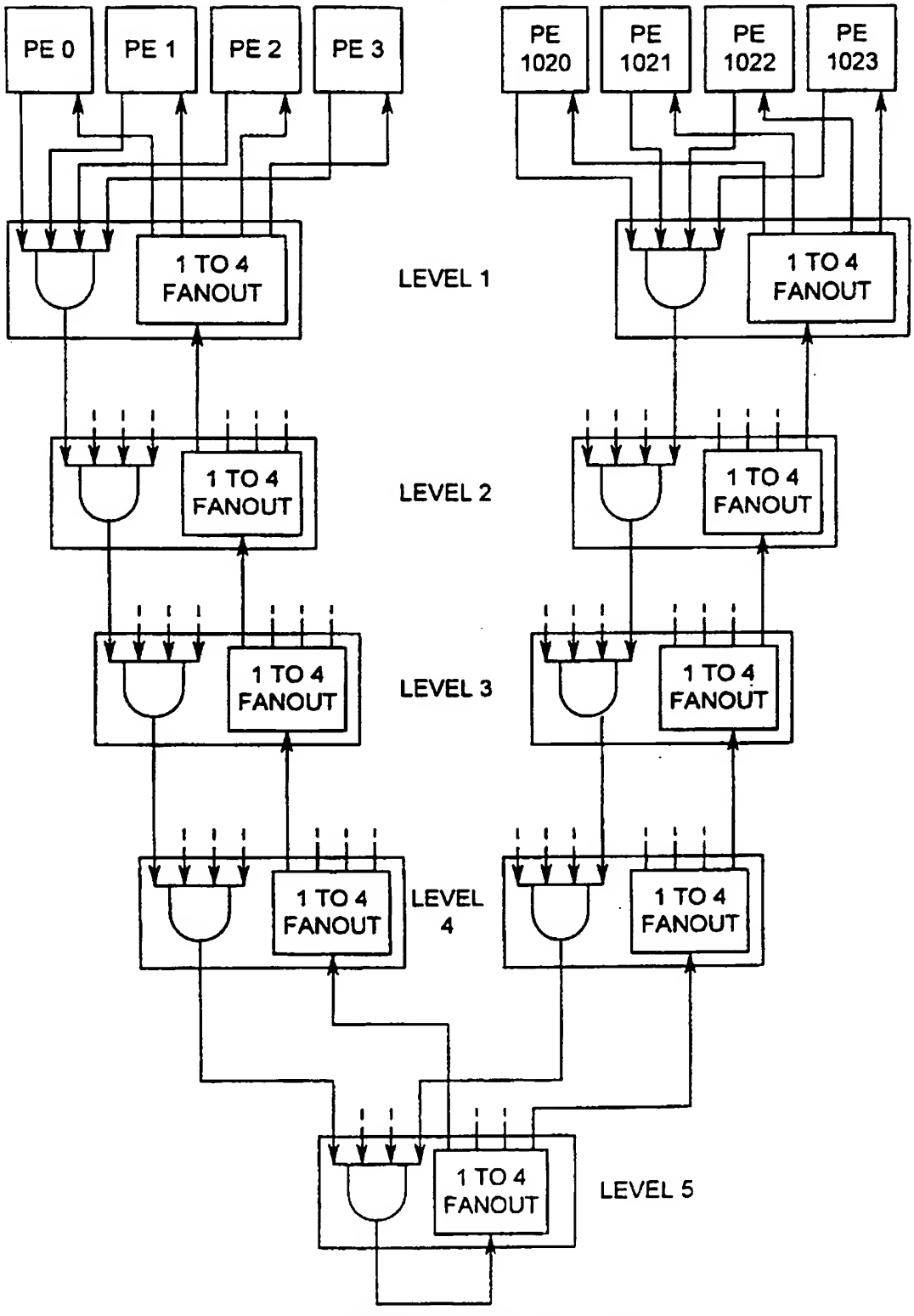

400
LEVEL 1
LEVEL 2
LEVEL 3
PE 7
PE 6
PE 5
PE 4
PE 3
PE 2
PE 1
PE 0
1 TO 2 FANOUT
1 TO 2 FANOUT
1 TO 2 FANOUT
1 TO 2 FANOUT
1 TO 2 FANOUT
1 TO 2 FANOUT
1 TO 2 FANOUT

Fig. 4

Fig. 5


PE 0
PE 1
PE 2
PE 3
PE 1020
PE 1021
PE 1022
PE 1023
1 TO 4 FANOUT
LEVEL 1
LEVEL 2
LEVEL 3
LEVEL 4
LEVEL 5

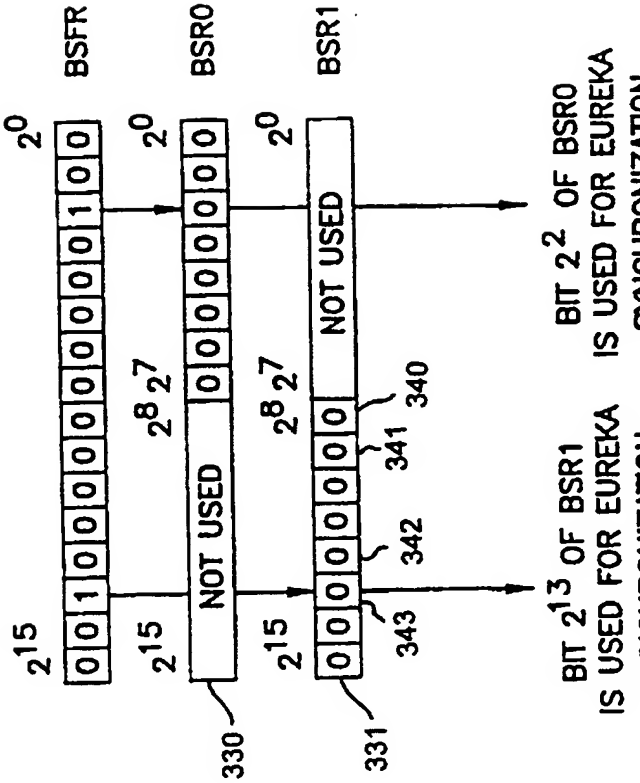

BSFR
BSR0
BSR1
2^0
2^{15}
2^8 2^7
NOT USED
330
331
340
341
342
343
BIT 2^{13} OF BSR1 IS USED FOR EUREKA SYNCHRONIZATION
BIT 2^2 OF BSR0 IS USED FOR EUREKA SYNCHRONIZATION

FIG. 6

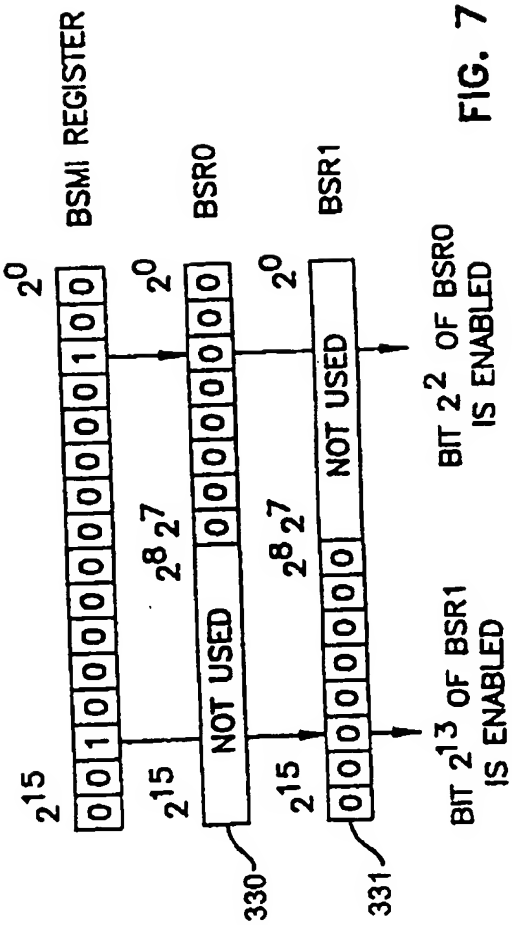

BSMI REGISTER
BSR0
BSR1
2^{15}
2^{0}
0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0
2^{8} 2^{7}
NOT USED
0 0 0 0 0 0 0 0
330
331
BIT 2^{13} OF BSR1 IS ENABLED
BIT 2^{2} OF BSR0 IS ENABLED

FIG. 7

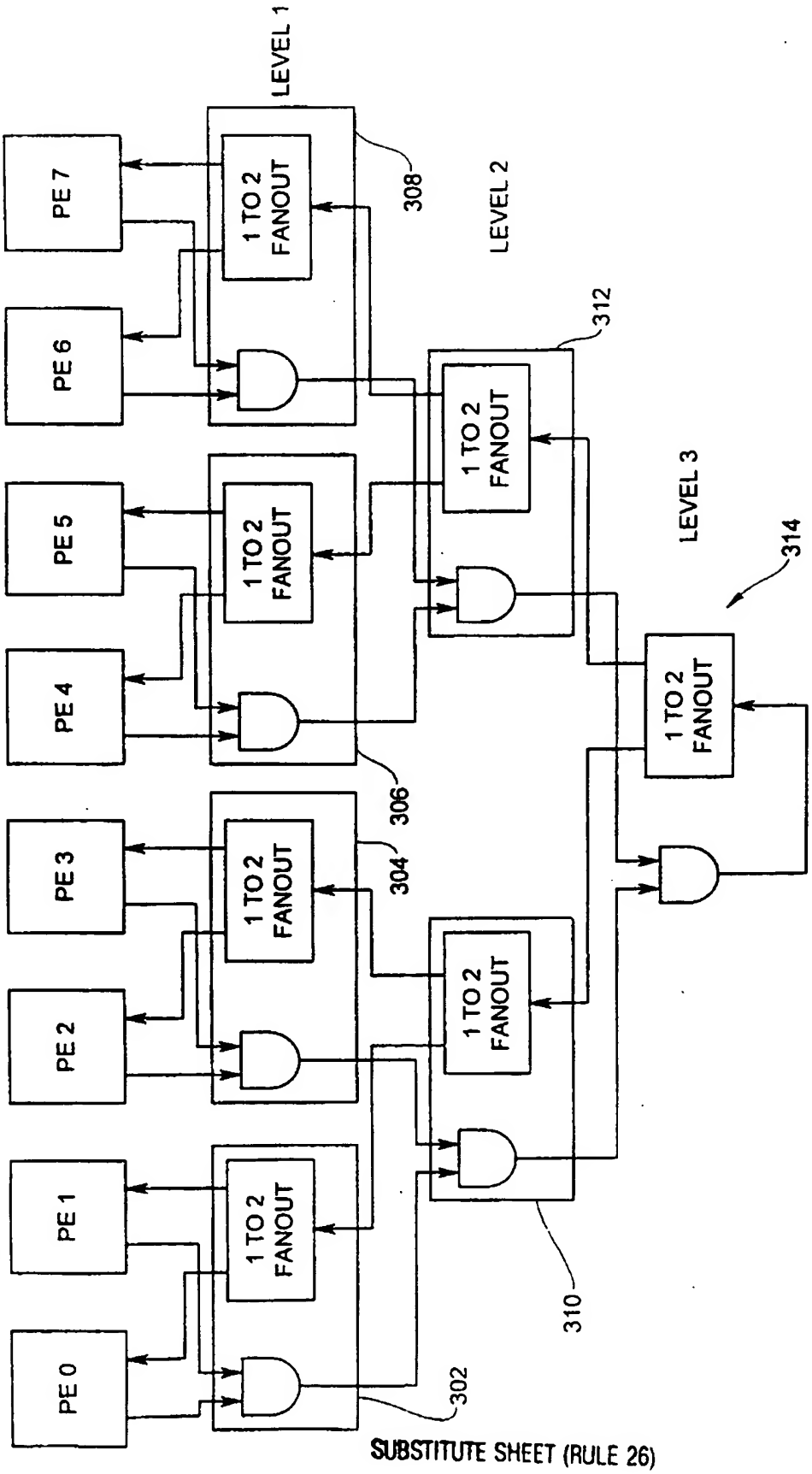

LEVEL 1
LEVEL 2
LEVEL 3
PE 0
PE 1
PE 2
PE 3
PE 4
PE 5
PE 6
PE 7
1 TO 2 FANOUT
302
304
306
308
310
312
314

Fig. 8

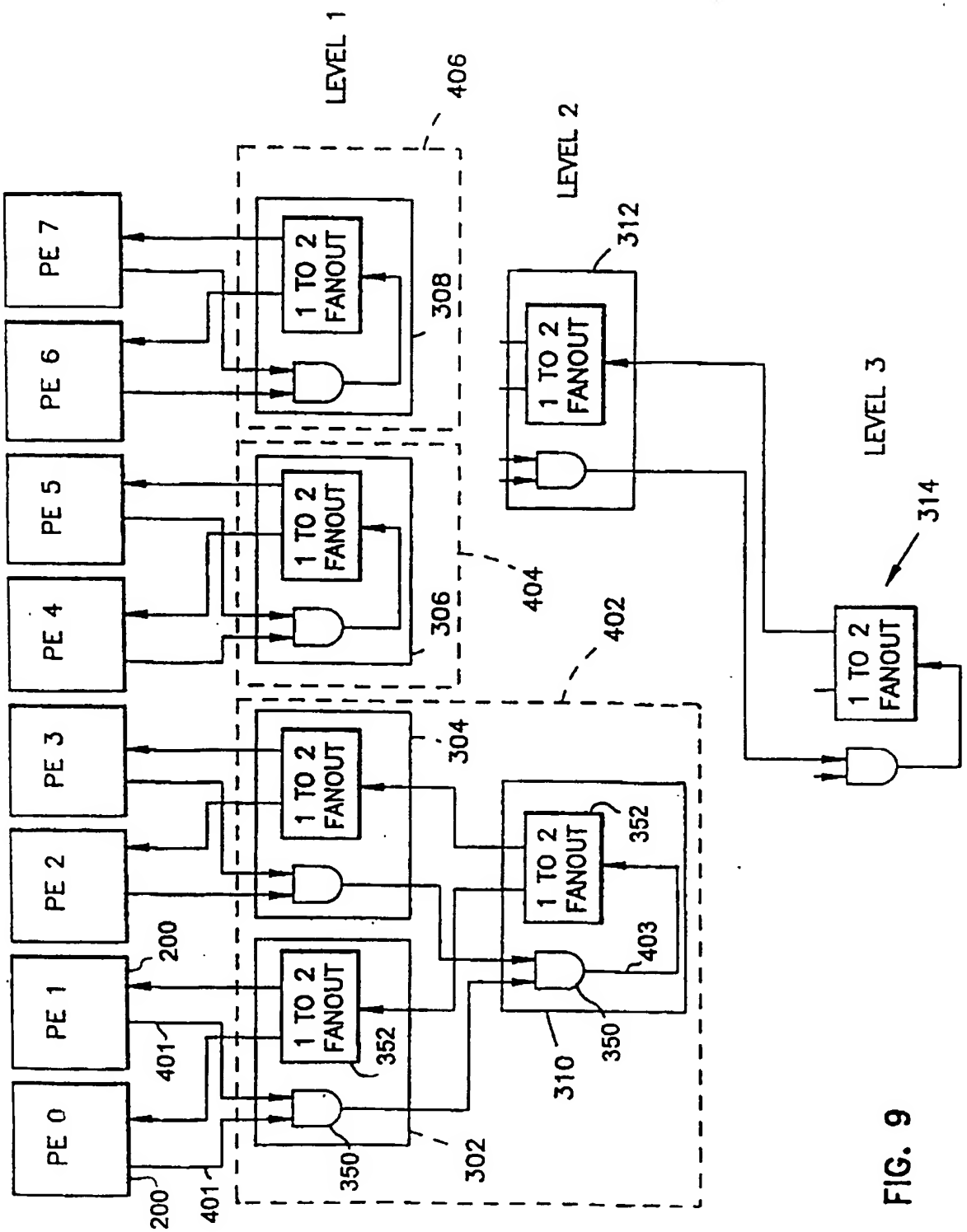

PE 0
PE 1
PE 2
PE 3
PE 4
PE 5
PE 6
PE 7
1 TO 2 FANOUT
LEVEL 1
LEVEL 2
LEVEL 3
200
401
350
352
302
304
306
308
310
312
314
402
403
404
406

FIG. 9

Fig. 10A

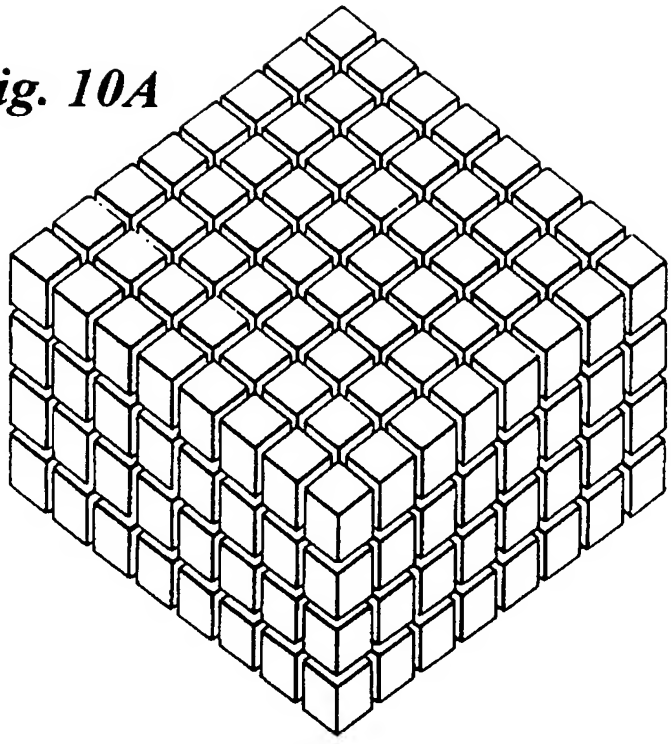


Fig. 10B

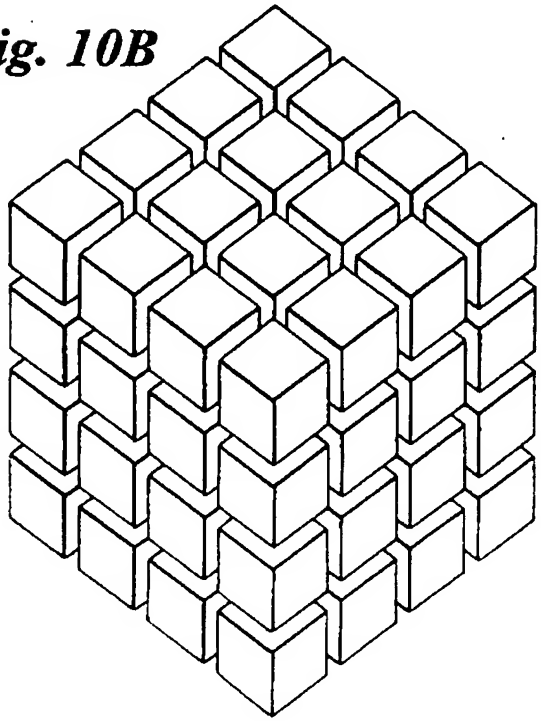


Fig. 10C

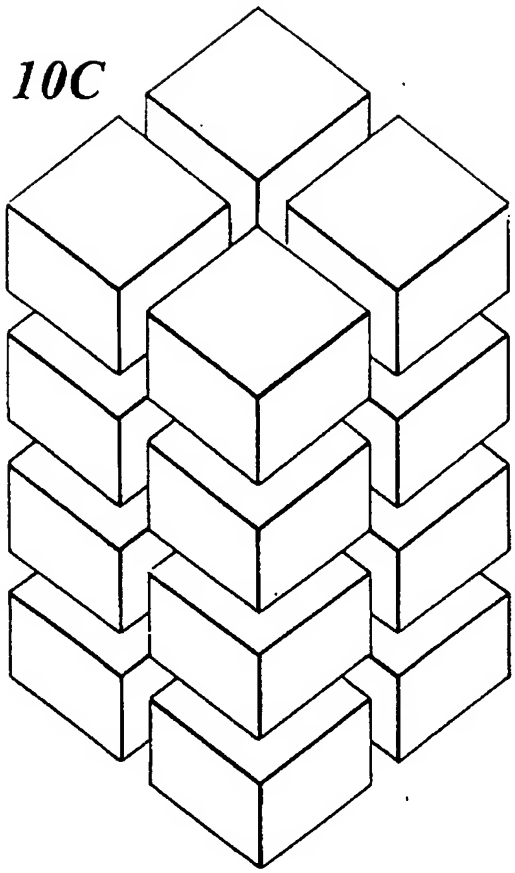


Fig. 10D

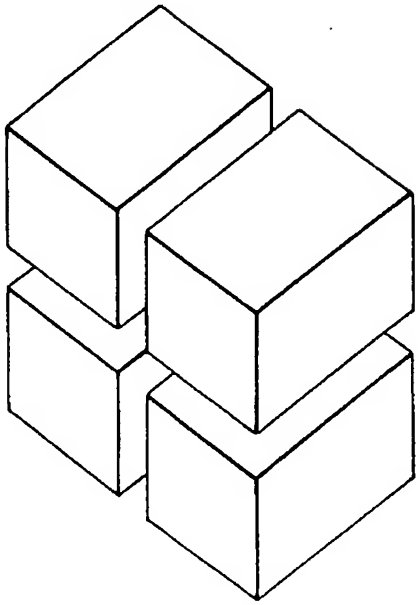
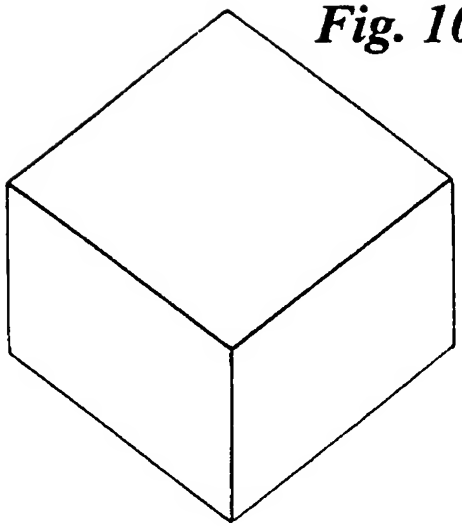


Fig. 10E



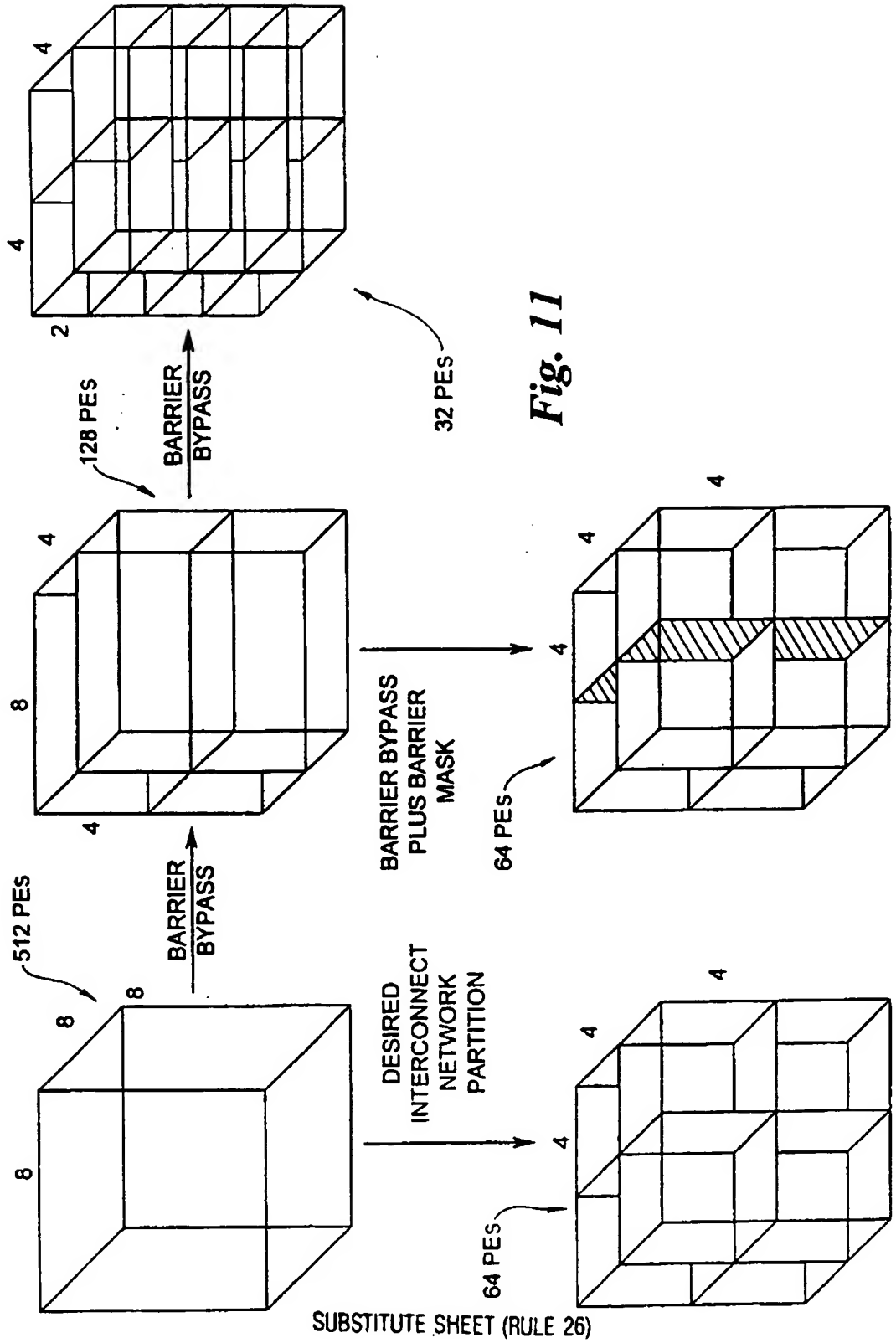

512 PEs
8
8
8
BARRIER BYPASS
128 PEs
4
4
8
4
BARRIER BYPASS
32 PEs
4
4
2
DESIRED INTERCONNECT NETWORK PARTITION
BARRIER BYPASS PLUS BARRIER MASK
64 PEs
4
4
4
64 PEs
4
4
4

Fig. 11

Fig. 12

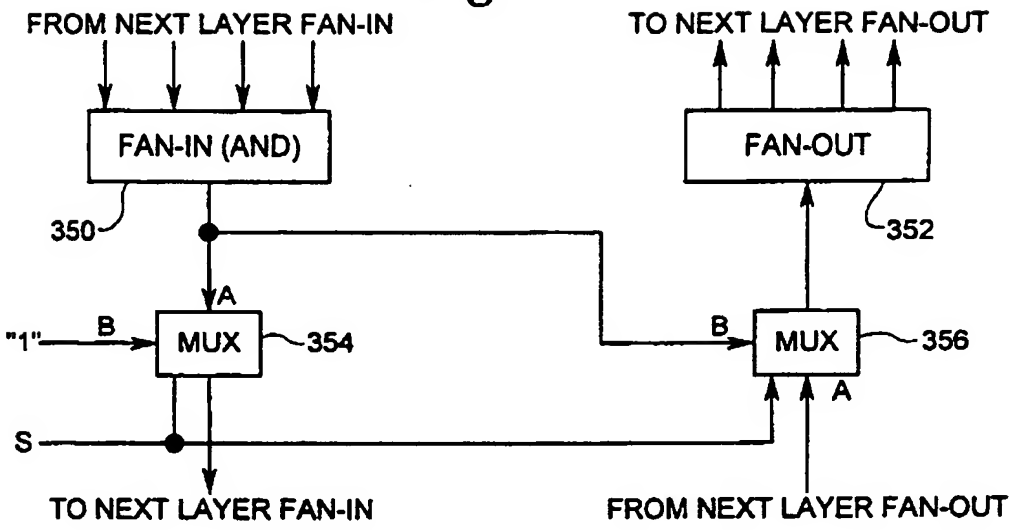
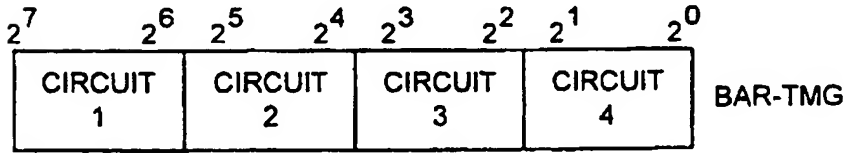

FROM NEXT LAYER FAN-IN
TO NEXT LAYER FAN-OUT
FAN-IN (AND)
FAN-OUT
350
352
A
"1"
B
MUX
354
B
MUX
356
A
S
TO NEXT LAYER FAN-IN
FROM NEXT LAYER FAN-OUT

Fig. 13


2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
CIRCUIT 1
CIRCUIT 2
CIRCUIT 3
CIRCUIT 4
BAR-TMG

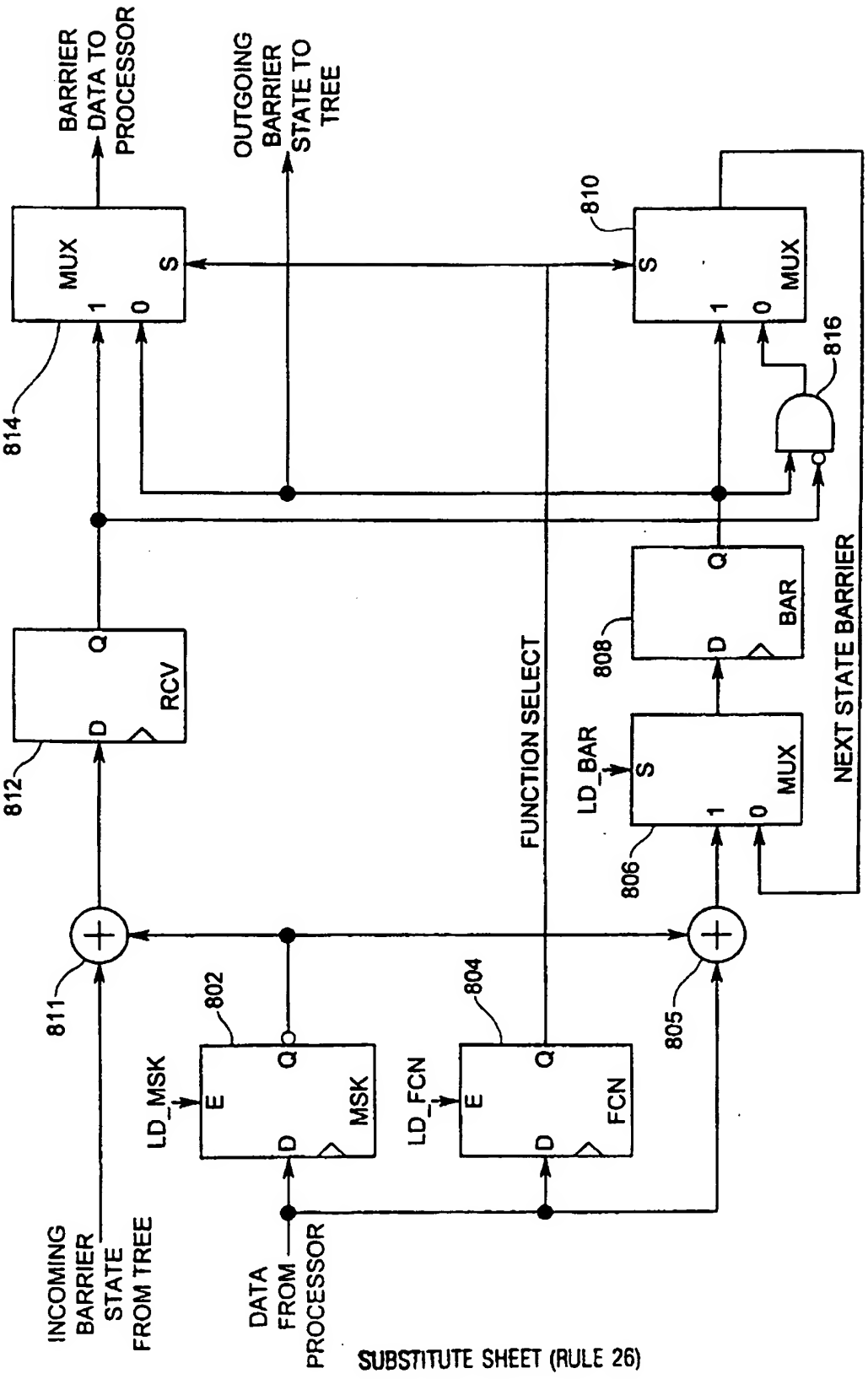

BARRIER DATA TO PROCESSOR
OUTGOING BARRIER STATE TO TREE
MUX
1
0
S
814
810
816
RCV
Q
D
812
BAR
808
LD_BAR
806
FUNCTION SELECT
NEXT STATE BARRIER
811
805
802
804
LD_MSK
LD_FCN
E
MSK
FCN
INCOMING BARRIER STATE FROM TREE
DATA FROM PROCESSOR

Fig. 14

INTERNATIONAL SEARCH REPORT

Internatr Application No
PCT/US 94/14067

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	INTERNATIONAL JOURNAL OF PARALLEL PROGRAMMING, vol.19, no.1, February 1990, NEW YORK US pages 53 - 73 R. GUPTA AND M. EPSTEIN 'High speed synchronization of processors using fuzzy barriers' see paragraph 7 ---	1,7,10
A	EP,A,0 353 819 (N. V. PHILIPS' GLOEILAMPENFABRIEKEN) 7 February 1990 see the whole document ---	1,7,10
A	EP,A,0 475 282 (HITACHI) 18 March 1992 see the whole document ---	1,7,10
	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
29 March 1995	04.04.95

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+ 31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+ 31-70) 340-3016	Michel, T

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Internat Application No
PCT/US 94/14067

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO,A,88 04810 (BELL COMMUNICATIONS RESEARCH) 30 June 1988 see the whole document	1,7,10

1

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

Internatne Application No
PCT/US 94/14067

Patent document cited in search report	Publication date	Patent family member(s)		Publication date
EP-A-0353819	07-02-90	JP-A-	2110763	23-04-90
EP-A-0475282	18-03-92	JP-A-	5002568	08-01-93
		US-A-	5361369	01-11-94
WO-A-8804810	30-06-88	US-A-	4833638	23-05-89
		CA-A-	1274020	11-09-90
		DE-A-	3782343	26-11-92
		EP-A,B	0337993	25-10-89
		JP-T-	2500306	01-02-90

TURNING EUREKA STEPS INTO CALCULATIONS IN AUTOMATIC PROGRAM SYNTHESIS *

A Bundy, A Smaill and J Hesketh

University of Edinburgh, UK

Abstract

We describe a technique called *middle-out reasoning* for the control of search in automatic theorem proving. We illustrate its use in the domain of automatic program synthesis. Programs can be synthesised from proofs that their logical specifications are satisfiable. Each proof step is also a program construction step. Unfortunately, a naive use of this technique requires a human or computer to produce proof steps which provide the essential structure of the desired program. It is hard to see the justification for these steps at the time that they are made; the reason for them emerges only later in the proof. Such proof steps are often call 'eureka' steps. Middle-out reasoning enables these eureka steps to be produced, automatically, as a side effect of non-eureka steps.

1 INTRODUCTION

We describe a technique called *middle-out reasoning* for the control of search during automatic program synthesis. Computer program synthesis is conducted by our Oyster program, Horn [1], which was based on the Cornell University, Nupri system, Constable *et al* [2]. To synthesise a program one provides a logical specification describing a relation, $spec(inputs, output)$, between the inputs and outputs of the proposed program. Oyster is then used as an interactive theorem prover to prove a conjecture of the form:

$$\forall inputs\ \exists output\ spec(inputs, output) \qquad (1)$$

in a logic based on Martin-Löf Intuitionist Type Theory[1], Martin-Löf [3]. This logic is *constructive*, *i.e.* the proof that an *output* exists for any combination of *inputs* must also show how to *construct* the *output* from the *inputs*. This construction recipe can be extracted from the proof and turned into a computer program.

Oyster proceeds by a process of backwards reasoning from the goal. Associated with each of its backwards proof steps is a program construction step. As the proof proceeds a functional/logic program, $prog(inputs)$, which is also expressed in the Type theory logic, is constructed. This program meets the specification:

$$\forall inputs.\ spec(inputs, prog(inputs))$$

There is a duality between proof steps and program construction steps, *e.g.* an inductive proof produces a recursive program. Since recursion is pervasive in functional/logic programs, we are particularly interested in proofs by mathematical induction. Such proofs present especially difficult problems of search control, namely in the choice of induction rule and induction variable(s).

To automate this process of program synthesis we have built a program Clam, van Harmelen [4], which guides Oyster in its search for a proof. Clam contains a collection of *tactics*. These are computer programs which apply Oyster rules, and hence direct its search. Clam analyses the conjecture to be proved and uses AI planning techniques to construct a *proof plan*, which is a tactic especially designed for the current conjecture. Further details of the Oyster-Clam system can be found in Bundy *et al* [5].

In Oyster's logic, a conjecture of form (1) is usually proved by eliminating its quantifiers at some stage, to form a goal of the form:

$$spec(inputs, prog(inputs)) \qquad (2)$$

where $prog(inputs)$ is called the *witness* of the existentially quantified variable *output*. This rather defeats the object of the exercise. As can be seen in goal (2), it is necessary for the Oyster user to *provide* the very program which Oyster is supposed to be synthesising. Even if the proof is first divided into cases and the quantifiers eliminated separately in each case, as in §3, the combination of the separate witnesses still defines the program.

The rest of the proof constitutes a mere *verification* that this program meets the specification, rather than a *synthesis* of the program. This elimination of an existential quantifier is an example of a *eureka* step, *i.e.* a step whose justification is not apparent at the time it is made. It becomes evident later in the proof, of course, when the verification succeeds. Eureka steps present a problem for both human and computer theorem proving. In either case it is hard to see how they might be thought of, *e.g.* what tactic might be implemented to make eureka steps.

*The research reported in this paper was supported by SERC grant GR/E/44596 and an SERC Senior Fellowship to the first author. We are grateful for discussions with the other members of the mathematical reasoning group at Edinburgh.

[1]However, in the interests of wider readability we have translated the Martin-Löf language into a more conventional notation in the examples given below.

Another example of a eureka step is the choice of appropriate induction variables and induction rule of inference. These will determine the forms of recursion to be used by the program.

2 PROGRAM SYNTHESIS BY THEOREM PROVING

To illustrate the technique of program synthesis and the eureka steps it requires, we will show how a program, *factors*(*x*), for factorising a positive integer, *x*, into its prime factors, can be synthesised from a proof of the prime factorisation theorem. The prime factorisation theorem can be expressed in English as:

> "For all positive integers, *x*, there is a list of prime numbers, *xl*, such that *x* equals the product of the numbers in *xl*."

It can be represented in our Type Theory as the formula:

$$\forall x{:}posint\ \exists xl{:}list(prime)\ prod(xl) = x \qquad (3)$$

where $X : T$ means X is an object of type T, *posint* is the type of positive natural numbers, *prime* is the type of prime numbers, and *list*(T) is the type of lists of objects of type T, e.g. $xl : list(prime)$ means *xl* is a list of primes. The function $prod : list(posint) \mapsto posint$ takes a list of numbers and returns their product, i.e.

$$\begin{aligned} prod(nil) &= 1 \\ prod(hd :: tl) &= hd \times prod(tl) \end{aligned}$$

where *nil* is the empty list and :: is the infix list constructor.

We can prove this theorem easily by using the *prime*× induction rule:

$$\frac{\vdash_\alpha P(1) \qquad p{:}primes, x'{:}posint, P(x') \vdash_\beta P(p \times x')}{\vdash_\gamma \forall x{:}posint.\ P(x)}$$

i.e. we prove the theorem for $x = 1$, then we assume it for $x = x'$ and prove it for $x = p \times x'$, where p is a prime number. The greek letters labelling the $\vdash$ symbols are the program fragments associated with the proofs of these sequents. The program construction step associated with this induction rule defines the γ program in terms of the α and β program fragments, as follows:

$$\begin{aligned} &\gamma(x) \Leftarrow \text{if}\ \ x = 1\ \ \text{then}\ \ \alpha \\ &\qquad \text{else}\ \beta(p, x') \text{ where } p = first(x) \wedge x' = rest(x) \end{aligned}$$

where *first*(*x*) is the smallest prime number that divides *x* and *rest*(*x*) is the quotient when *x* is divided by *first*(*x*). In this case *factors*(*x*) = $\gamma(x)$. So the decision to use *prime*× induction ensures that the program will use a dual form of recursion. Deciding to use this esoteric form of induction is our first eureka step.

The base case of the induction is:

$$\vdash_\alpha \ \exists xl{:}list(prime)\ prod(xl) = 1 \qquad (4)$$

and the step case is:

$$\begin{aligned} &p{:}prime, \\ &x'{:}posint, \\ &\exists xl{:}list(prime)\ prod(xl) = x' \ \vdash_\beta \\ &\qquad \exists xl{:}list(prime)\ prod(xl) = p \times x' \end{aligned} \qquad (5)$$

The base case is readily proved by eliminating the existential quantifier in (4) and introducing the witness *nil* for *xl*. This also instantiates α to *nil*. Similarly, the step case is proved by first eliminating the existential quantifier in the induction hypothesis part of (5) with witness *xl'* and then the one in the induction conclusion with witness $p :: xl'$. Oyster is able to work out that *xl'* is *factors*(x'), so it instantiates β to $p :: factors(x')$. The program is now:

$$\begin{aligned} &factors(x) \Leftarrow \\ &\quad \text{if}\ \ x = 1\ \ \text{then}\ nil \\ &\quad \text{else}\ p :: factors(x') \\ &\qquad \text{where}\ p = first(x) \wedge x' = rest(x) \end{aligned}$$

The program is now fully synthesised.

The witness introduced by the elimination of the existential quantifier in the induction hypothesis is standard, but the witnesses introduced by the elimination of the other two quantifiers constitute eureka steps. Note how these two witnesses are the values of the function *factors*(*x*) in the base and step cases of the recursion.

3 THE *RIPPLE_OUT* TACTIC

The remaining parts of the proof will provide a verification that our choices of induction rule and existential witnesses yield a program that meets the specification. The remaining part of the step case is to prove:

$$\begin{aligned} &p{:}prime, \\ &x'{:}posint, \\ &xl'{:}list(prime) \\ &prod(xl') = x' \ \vdash_\beta \ \boxed{p ::} xl'{:}list(prime) \wedge \\ &\qquad prod(\boxed{p ::} xl') = \boxed{p \times} x' \end{aligned} \qquad (6)$$

Note that the existential quantifiers have all been eliminated and the existential variables replaced by their witnesses. The witness of the existential variable in the induction conclusion, $p :: xl'$ must be shown to have the right type, *list*(*prime*), so the subgoal, $p :: xl' : list(prime)$, becomes part of the induction conclusion.

Three sub-expressions of the induction conclusion have been placed in boxes. These are examples of *wave fronts*. *Wave fronts* are those sub-expressions of the induction conclusion in which it differs from the induction hypothesis. The Clam system contains a tactic called *ripple_out* whose task is to make the induction hypothesis appear as a sub-expression of the induction conclusion. It works by rewriting the induction conclusion to move the wave fronts outwards from their original deeply nested positions. The

rules used by *ripple_out* are called *wave rules*. Wave rules are rewrite rules of the form[9]:

$$F(\boxed{S_1(}U_1\boxed{)},\ldots,\boxed{S_n(}U_n\boxed{)}) \Rightarrow \boxed{T(}F(U_1,\ldots,U_n)\boxed{)}$$

where F, T and the S_i are terms with distinguished arguments and T may be empty, but F and the S_i must not be. The S_i are old wave fronts and T is the new wave front. Application of a wave rule ripples some wave fronts out by one stage. Repeated application ripples them all to the outside of the induction conclusion.

The wave rules required for this proof are:

$$\boxed{hd ::}\,tl:list(type) \Rightarrow \boxed{hd:type\wedge}\,tl:list(type) \quad (7)$$

$$prod(\boxed{hd ::}\,tl) \Rightarrow \boxed{hd\times}\,prod(tl) \quad (8)$$

$$\boxed{u\times}\,v = \boxed{u\times}\,w \Rightarrow v = w \quad (9)$$

Applying these wave rules to the induction conclusion of (6) rewrites the step case as follows. After application of rule 7 to the first wave front, Clam derives:

$$\begin{array}{l} p:prime, \\ s':posint, \\ sl':list(prime) \\ prod(sl') = s' \end{array} \vdash_\rho \boxed{p:prime\wedge}\,sl':list(prime) \wedge prod(\boxed{p ::}\,sl') = \boxed{p\times}\,s'$$

After application of rule 8 to the second wave front, it derives:

$$\begin{array}{l} p:prime, \\ s':posint, \\ sl':list(prime) \\ prod(sl') = s' \end{array} \vdash_\rho \boxed{p:prime\wedge}\,sl':list(prime) \wedge \boxed{p\times}\,prod(sl') = \boxed{p\times}\,s'$$

And after application of rule 9, simultaneously, to the second and third wave fronts, it derives:

$$\begin{array}{l} p:prime, \\ s':posint, \\ sl':list(prime) \\ prod(sl') = s' \end{array} \vdash_\rho \boxed{p:prime\wedge}\,sl':list(prime) \wedge prod(sl') = s'$$

Figures 1 and 2 illustrate this rippling-out process graphically by showing two of the stages that the induction conclusion goes through.

After these three rule applications each of the three conjuncts of the induction conclusion is identical to one of the induction hypotheses. The induction hypothesis can then be used to prove the induction conclusion. The Clam system has a tactic called *fertilisation* whose task is to match the induction hypothesis against sub-expressions of the induction conclusion and replace any such sub-expressions by *true*. Doing this completes the proof of the step case in this example.

A fuller explanation of *ripple_out* can be found in [6].

[9] This is not the most general form that wave rules can take, but is sufficient for the examples in this paper. A more general form is given in Bundy *et al* [6].

4 MIDDLE-OUT REASONING

We can think about the eureka steps described in §3 in the following way. The choices of induction scheme and existential witnesses are actually made in such a way as to allow the rest of the proof to proceed successfully. In particular, the eureka choices will enable the subsequent *ripple_out* tactic to succeed. However, since the eureka steps are made before *ripple_out* is applied, this is not immediately obvious.

This observation suggests a way to automate the eureka decisions, namely: postpone making the eureka steps and apply *ripple_out first*; then integrate the eureka steps into the rippling as required to enable it to continue. Effectively we do the middle of the proof first. In the process we calculate what form the beginning of the proof should take to make the middle of it succeed. We call this strategy *middle-out reasoning*.

4.1 Rippling Under Existential Quantifiers

We consider first the problem of choosing witnesses for existentially quantified variables. Our solution is not to eliminate the existential quantifiers at all, but to modify the rewriting procedure so that rippling can be carried out under existential quantifiers. To illustrate this process, consider again the proof of the prime factorisation theorem. We return to the step case of the proof, just after the application of induction.

$$\begin{array}{l} p:prime, \\ s':posint, \\ \exists sl':list(prime)\ prod(sl') = s' \end{array} \vdash_\rho \exists\boxed{sl}:list(prime)\ prod(\boxed{sl}) = \boxed{p\times}\,s' \quad (10)$$

This time we have marked in the wave fronts *before* eliminating the existential quantifier. Recall that the witness for sl in the induction conclusion, $p :: sl'$, contained a wave front, $\boxed{p ::}$ (see (6)). This justifies our marking this existential variable as a wave front.

This remark generalises to all existentially quantified variables in induction conclusion. Their witnesses will be the value of the synthesised program in the step case of the recursion, *i.e.* some function of the value of the program when it is called recursively. When this function is not the identity function, then it will be a wave front. So all existential variables in induction conclusions are potential wave fronts. Note that the occurrence of the existential variable in the quantifier declaration is marked as a wave front, as well as all occurrences in the body of the formula. This is because the quantifier declaration also contains a

type declaration, which will require rippling with wave rules for types, like (7) above.

We now proceed to apply *ripple_out*, using the same wave rules as in §3, but without first removing the $\exists sl : list(prime)$. Our modified rewriting procedure has the freedom to instantiate existential variables to compound terms of the same type. This is because in the unmodified procedure these compound terms could have been introduced as witnesses before *ripple_out* was applied. To take advantage of this possibility the modified rewriting procedure matches the left hand side of the rewrite rule with expressions in the goal, treating existential variables as free variables that can be instantiated. When an existential variable is instantiated to a witness of the same type, the existential quantifier that governs it will no longer be required, and should be deleted. However, a new existential quantifier will be required for each variable contained in the witness. To assist with checking that the witness has the required type and with the calculation of the types of the new existential variables, it helps if the wave fronts in the quantifier/type declarations are rippled before those in the body of the formulae.

Consider, for instance, the application of wave rule (7) to the first wave front in (10).

$$\boxed{\exists hd : prime}\ \exists tl : list(prime)\ prod(\boxed{hd :: tl}) = \boxed{p \times}s'$$

This instantiates sl, and hence β, to $hd :: tl$. The existential quantifier governing sl is replaced by two existential quantifiers: one for hd and one for tl. Note how the types of these new existential variables are provided by the right hand side of rule (7). This use of a type rule also guarantees that the witness has the right type. Note also how the second occurrence of sl, in the body of the formula, has been instantiated to $hd :: tl$.

The witness to the existential variable, sl, has now been determined. It was not necessary to do this as a ground step. The appropriate witness was calculated by the matching routine during the application of *ripple_out*. No search was required for this. The wave front around sl had to be rippled out at some time (in fact, it is best to ripple it out first). Rule (7) was the only matching wave rule because of the restriction imposed by the sub-expression $list(prime)$, the type of sl.

We now apply wave rule (8) to the second wave front, yielding:

$$\boxed{\exists hd : prime}\ \exists tl : list(prime)\ \boxed{hd \times}\ prod(tl) = \boxed{p \times}s'$$

This application does not instantiate any existential variables, so does not require any changes in the existential quantifiers.

We can now apply multi-wave rule (9), simultaneously, to the second and third wave fronts, yielding:

$$\exists tl : list(prime)\ prod(tl) = s'$$

This rewrite instantiates hd to p. This instantiation is only possible because p is a term with the same type as hd. Since hd has been instantiated its existential quantifier must be removed. Since p is a constant, and therefore contains no new variables, no new existential quantifiers are introduced. This has the side effect of removing the first wave front. The remaining formula matches the induction hypothesis. So fertilization removes it, instantiating tl to sl', and hence sl and β to $p :: sl'$, as required.

A similar process can be carried out in the base case. The role of *ripple_out* being played by the Clam tactic, *base*, which rewrites formulae using the base cases of recursive definitions.

4.2 Using Meta-Variables for Induction Terms

We now consider the problem of choosing induction rules. Our solution is make a 'least commitment' induction step by using a schematic induction rule in which the induction term is a meta-variable. We then allow this meta-variable to become instantiated during the subsequent rippling out. Since meta-variables are not allowed in Oyster's logic, this reasoning is handled within the Clam planner. Clam applies *ripple_out* and works out what induction rule is required. It then instructs Oyster to apply the appropriate induction rule and then ripples the wave fronts that induction creates.

To illustrate this we return again to the step case of the prime factorisation proof, just after the application of induction, *i.e.*

$$\begin{array}{l} s' : posint, \\ \exists sl : list(prime)\ prod(sl) = s' \vdash_\rho \\ \quad \exists \boxed{sl} : list(prime)\ prod(\boxed{sl}) = \boxed{X} \end{array}$$

but instead of the induction term, $p \times s'$, we have used the meta-variable, X, which stands for some term containing s', *i.e.* we have replaced all occurrences of s', in the induction conclusion, by X. All universally quantified variables are candidate induction variables, but s is the only candidate in this example. In general, we must try replacing each universal variable, in turn, by such a meta-variable, to see which replacements permit *ripple_out* to succeed. Note that the declaration of p has been omitted from the hypothesis, since we do not yet know what parameters might be introduced by the induction.

Rippling-out then proceeds as in §4.1 until we get to the formula:

$$\boxed{\exists hd : prime}\ \exists tl : list(prime)\ \boxed{hd \times}\ prod(tl) = \boxed{X}$$

Now rule (9) is the only wave rule that applies to the second wave front. Applying it produces:

$$\boxed{\exists hd : prime}\ \exists tl : list(prime)\ prod(tl) = X'$$

where X is instantiated to $hd \times X'$. *fertilization* applies to this, instantiating X' to s' and leaving the residue:

$$\exists hd : prime\ true$$

which is provable provided hd is witnessed by a prime number.

We have thus established that the *ripple_out* tactic will succeed provided that the induction term is $Ad \times s'$, where $Ad : prime$. Induction rules are indexed in Clam by their induction terms, so this information enables the systems to recover the *prime*× induction rule and use it retrospectively. Notice how the appropriate induction rule was chosen as a side-effect of the *ripple_out* tactic.

5 CONCLUSION

In this paper we have described a technique, based on theorem proving, for synthesising programs from logical specifications. We have seen that a naive use of this technique requires eureka steps. In §2 we saw that it was necessary to produce an appropriate induction rule and appropriate existential witnesses 'out of the blue'. Furthermore, these eureka steps provided all the essential structure of the synthesised program, leaving only the verification that this program met the specification. These eureka steps constitute a barrier to the use of the program synthesis technique. This is true whether we want to use it totally automatically, with a computer providing the eureka steps, or semi-automatically, with a human providing them.

We have described a way of finessing the eureka steps, so that program synthesis can be automated. We draw on previous work in which we automated the verification part of the proof. In particular, our *ripple_out* tactic is highly successful in automatically guiding the proving of the induction conclusion from the induction hypothesis. We have arranged the proof construction so that this middle part of the proof is done first. The eureka steps emerge as a side effect of *ripple_out* — they are made in such a way as will permit *ripple_out* to continue. We call this technique *middle-out reasoning*. We are currently implementing middle-out reasoning within the Oyster-Clam system.

References

[1] C. Horn. *The Nurpri Proof Development System.* Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurpri has been renamed Oyster.

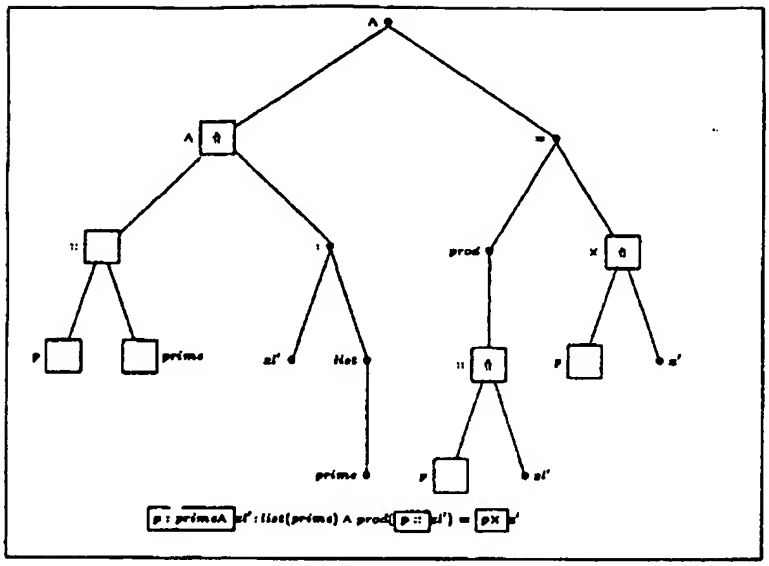
[2] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[3] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1979. Published by North Holland, Amsterdam. 1982.

[4] F. van Harmelen. *The CLAM Proof Planner, User Manual and Programmer Manual.* Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.

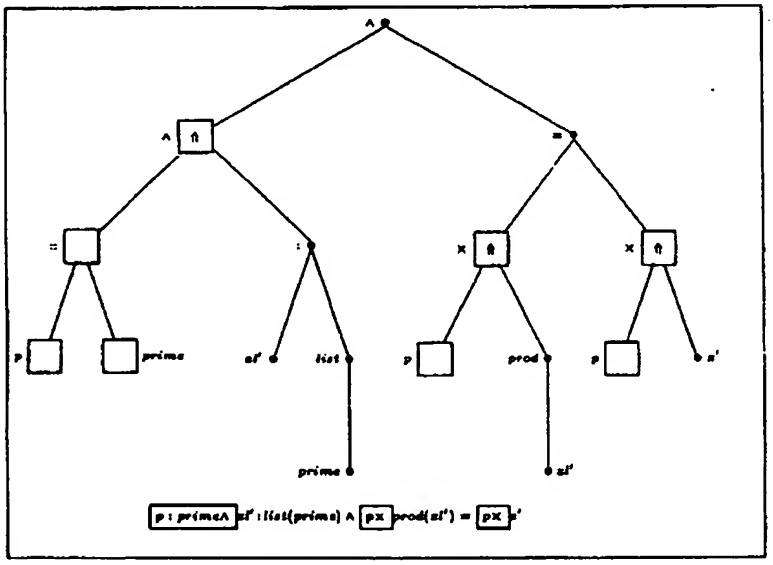
[5] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. *Experiments with Proof Plans for Induction.* Research Paper 413, Dept. of Artificial Intelligence, Edinburgh, 1988. To appear in JAR.

[6] A. Bundy, van Harmelen. F., and A. Smaill. *Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs.* Research Paper forthcoming, Dept. of Artificial Intelligence, Edinburgh, 1989. Submitted to CADE-10.



The induction conclusion is shown after the application of one wave rule both as an expression tree and as a formula. Each non terminal node of an expression tree is labelled by a function or predicate symbol. Each terminal node is labelled by a constant or variable. A function or predicate of arity n has n descendant subtrees: one for each of its arguments. Wave fronts are indicated by square nodes and all other nodes by dots. The ⇑ in the top square node indicates the direction of rippling out.

Figure 1: Induction Conclusion After One Ripple



The induction conclusion is shown after the application of a further wave rule. Note that this tree differs from the first one only in the square nodes. Each ripple moves the square nodes higher up the tree.

Figure 2: The Induction Conclusion After Two Ripples

Non-Uniformities Introduced by Virtual Channel Deadlock Prevention

Kevin Bolding

July 21, 1992

Abstract

A common scheme for preventing deadlock in networks is the virtual channel method of Dally and Seitz [DS87]. Due to the nature of this scheme, an otherwise completely uniform network will have non-uniformities introduced into it. The variations introduce several effects, ranging from limitations on overall network performance to differences in observed network characteristics from node to node and from message to message.

1 Introduction

A useful multicomputer network must be both efficient and reliable. A key component of reliability is freedom from deadlock. Many schemes have been introduced which prevent deadlock in general networks, at varying costs in terms of additional buffers and complexity. A particularly interesting scheme is presented by Dally and Seitz [DS87] which relies on *virtual channels* to break dependency cycles in a network. The general scheme is to note where cycles may exist in a network and routing scheme and then to add virtual channels in order to transform the circular dependencies into spirals which are, by nature, acyclic.

While *open-ended* k-ary d-cubes (*e.g.* meshes and hypercubes) can be made deadlock-free by restricting routing to be oblivious and dimension ordered, this cannot be extended to k-ary d-cubes where the edges are "wrapped around" (*e.g.* tori). A scheme presented in [DS87] achieves deadlock freedom in wrapped-around networks by dividing each physical channel into two virtual channels and restricting the routing to be dimension ordered. A further restriction prevents deadlock: a message uses the "high" set of virtual channels if the number of the destination address is higher than the current node, and the "low" set of virtual channels if the destination address is less than or equal to the current node address. This prevents deadlock by choosing a *terminal* node at which dependencies in a particular set of virtual channels must terminate, thus preventing cyclic dependencies from occurring.

To achieve maximum performance in a particular network, each of the resources in the network should be loaded equally. If one component reaches saturation before the rest, the network will tend to slow to the load which can be handled by the saturated component.[1] In a sense, this node has become the "weakest link" in a chain, and the strength of the entire chain is thus diminished. This, for example, is responsible for early saturation in mesh networks where there is extra congestion in the center of the network. A torus network exhibits *vertex transitivity* which means that there are no observable differences between any two nodes in the network. Thus, torus networks should not have the problem of congestion in the center because there is

[1] This is true for uniform random traffic, but with non-uniform traffic local saturations may not spread to cover the entire network.

no distinction between nodes in the center and nodes along the edges of a torus. However, when the virtual channel deadlock scheme is implemented, the routing mechanism is perturbed in a manner which re-introduces non-uniformities by restricting which messages may use particular buffers.

2 Uneven buffer utilization

Although the virtual channel scheme successfully prevents deadlock, it does so by the addition of underutilized buffers. Furthermore, the effective buffer space available to each node varies according to the node's position in the network, creating non-uniformities in the network.

Consider the set P of all paths between any two nodes allowed by the network and routing algorithm. Each channel c in the network is part of a set of paths p_c which use c. Also, each virtual channel c_v of channel c has a set of paths p_{c_v} which use c_v. We know that $|p_{c_v}| \leq |p_c|$ since no virtual channel can have a greater number of paths through it than the channel to which it is attached. Now define the virtual channel utilization $u_{c_v} = \frac{|p_{c_v}|}{|p_c|}$ for each virtual channel. ($\sum_{c_v} u_{c_v} = 1$) Thus, u_{c_v} is the fraction of the paths through channel c which use virtual channel c_v.

Now, since each channel is represented by multiple virtual channels, the most effective use of the buffer space provided by the virtual channels is for each virtual channel to have the same fraction of the total traffic. If one virtual channel has more traffic than the others, it becomes the bottleneck for the channel. Thus we define the effective buffer size $B = \frac{1}{\max_{c_v}(u_{c_v})}$. If all the buffers are utilized equally, B will equal the number of buffers; if only one buffer is utilized, B will equal 1.

2.1 Unidirectional channels

Consider a unidirectional ring network under random traffic. This can be thought of as a "slice" of a torus, and since, in dimension-order routing, interactions between dimensions are minimal, the analysis extends to multi-dimensional tori as well. For channel c, $|p_c| = \sum_{i=0}^{N-1} i$.

The scheme proposed in [DS87] routes messages on the high virtual channels if the node number of the source node is less than the destination node, and the low virtual channels otherwise. This results in the "terminal" node being node zero. The number of paths through the high virtual channels is $|p_{c_H}| = \sum_{i=0}^{j} i$ for node j. The number of paths through the low virtual channels is $|p_{c_L}| = \sum_{i=j+1}^{N-1} i$. Thus, for a unidirectional ring, the effective buffer size is:

$$B(j) = \frac{\sum_{i=0}^{N-1} i}{\max\left(\sum_{i=0}^{j} i, \sum_{i=j+1}^{N-1} i\right)}$$

The scheme implemented in designing the torus routing chip [DS86] injects all messages (except from node zero) on the low virtual channels. Messages are routed on the low channels until they cross node zero, when they switch to the high virtual channels and continue on them until they reach their destination. This also results in the terminal node being node zero. The number of paths through the high virtual channels is $|p_{c_H}| = \sum_{i=N-j}^{N-1} i$ for node j. The number of paths through the low virtual channels is $|p_{c_L}| = \sum_{i=0}^{N-j} i$. Thus, for a unidirectional ring,

Node	Dest < Current			Cross 0		
	$\lvert p_{c_H} \rvert$	$\lvert p_{c_L} \rvert$	B	$\lvert p_{c_H} \rvert$	$\lvert p_{c_L} \rvert$	B
0	0	120	1.000	0	120	1.000
1	1	119	1.008	15	105	1.143
2	3	117	1.026	29	91	1.319
3	6	114	1.053	42	78	1.538
4	10	110	1.091	54	66	1.818
5	15	105	1.143	65	55	1.846
6	21	99	1.212	75	45	1.600
7	28	92	1.304	84	36	1.429
8	36	84	1.429	92	28	1.304
9	45	75	1.600	99	21	1.212
10	55	65	1.846	105	15	1.143
11	66	54	1.818	110	10	1.091
12	78	42	1.538	114	6	1.053
13	91	29	1.319	117	3	1.026
14	105	15	1.143	119	1	1.008
15	120	0	1.000	120	0	1.000

Table 1: Effective buffer size for a 16-node unidirectional ring.

the effective buffer size is:

$$B(j) = \frac{\sum_{i=0}^{N-1} i}{\max\left(\sum_{i=N-j}^{N-1} i, \sum_{i=0}^{N-j} i\right)}$$

The number of paths through each set of virtual channels, as well as effective buffer size, are shown in Table 1.

2.2 Bidirectional channels

For a bidirectional ring, the number of paths through any channel is different because of the shorter path lengths. For channel c, $|p_c| = \sum_{i=0}^{N/2} i$.

For the "dest < current" scheme, the number of paths through the high virtual channels is $|p_{c_H}| = \sum_{i=0}^{j-(N-1)/2} i$ for node j. The number of paths through the low virtual channels is $|p_{c_L}| = \sum_{i=j-(N-1)/2+1}^{N/2} i$.[2] Thus, for a bidirectional ring, the effective buffer size is:

$$B(j) = \frac{\sum_{i=0}^{\frac{N}{2}} i}{\max\left(\sum_{i=0}^{j-\frac{N-1}{2}} i, \sum_{i=j-\frac{N-1}{2}+1}^{\frac{N}{2}} i\right)}$$

[2]We define the sum where the lower bound is negative to be the sum from a lower bound of zero.

Node	Dest < Current			Cross 0		
	$\lvert p_{c_H} \rvert$	$\lvert p_{c_L} \rvert$	B	$\lvert p_{c_H} \rvert$	$\lvert p_{c_L} \rvert$	B
0	0	36	1.000	0	36	1.000
1	0	36	1.000	8	28	1.286
2	0	36	1.000	15	21	1.714
3	0	36	1.000	21	15	1.714
4	0	36	1.000	26	10	1.385
5	0	36	1.000	30	6	1.200
6	0	36	1.000	33	3	1.091
7	0	36	1.000	35	1	1.029
8	1	35	1.029	36	0	1.000
9	3	33	1.091	36	0	1.000
10	6	30	1.200	36	0	1.000
11	10	26	1.385	36	0	1.000
12	15	21	1.714	36	0	1.000
13	21	15	1.714	36	0	1.000
14	28	8	1.286	36	0	1.000
15	36	0	1.000	36	0	1.000

Table 2: Effective buffer size for a 16-node bidirectional ring.

For the "crossing node zero" scheme, the number of paths through the high virtual channels is $|p_{c_H}| = \sum_{i=N/2-j+1}^{N/2} i$ for node j. The number of paths through the low virtual channels is $|p_{c_L}| = \sum_{i=0}^{N/2-j} i$. Thus, for a bidirectional ring, the effective buffer size is:

$$B(j) = \frac{\sum_{i=0}^{\frac{N}{2}} i}{\max\left(\sum_{i=\frac{N}{2}-j+1}^{\frac{N}{2}} i, \sum_{i=0}^{\frac{N}{2}-j} i\right)}$$

The number of paths through each set of virtual channels, as well as effective buffer size, are shown in Table 2.

3 Significance

The effect of uneven buffer utilization on performance varies with the amount of buffering in the network. If wormhole routing is implemented with minimal (i.e., single flit) buffering per virtual channel, then performance will not be effected tremendously because the buffers are not playing a critical role in achieving performance. On the other hand, increasing buffering in a network can dramatically boost its performance. Thus, it makes sense to use all of the buffer space available. We see that, in a unidirectional network, the nodes on either side of the "terminal" node have only one usable buffer. In a bidirectional network, this is true for fully one-half of the nodes. Since these nodes become "weakest link" nodes, the network will perform roughly the same as a network which has only one buffer per channel, despite the presence of twice as much buffering capability.

Moreover, the amount of effective buffer space at each node varies from node to node. This results in a very uneven distribution of usable resources within a network and has several negative effects on performance. First of all, the latency which a message experiences depends on which portion of the network it travels through, even under a uniform random load. Ideally, a network will provide uniform performance under a uniform load so that users will not have to worry about being dealt "bad" nodes. Secondly, nodes which have more effective buffer space will have a greater number of opportunities to inject messages into the network, potentially preventing other nodes from injecting messages and resulting in starvation for those nodes. These effects combined to reduce the expected performance of the oblivious torus router described in [BS92].

A third non-uniformity exists which may present different views of network contention to messages which have different distances between their sources and destinations. For the "Dest < Current" scheme, the virtual channel a message uses depends on whether or not the message will *eventually* cross the terminal node (node zero). Consider two messages from the same source which are traveling the same direction at some node common to both of their paths from their source to their destinations. If one of the messages has a destination which is numbered lower than the current node, while the other has a destination which is numbered higher than the current node, the two messages will use different virtual channels. Since they use different virtual channels, they will compete with different messages for buffer space and thus face different levels of congestion. However, from a local point of view, the two messages have arrived on the same channel and wish to leave on the same channel, and should not be distinguishable. Thus, the deadlock prevention mechanism discriminates between messages which should be treated equivalently in a uniform routing strategy. In particular, messages with longer paths from source to destination are more likely to cross the terminal node and be routed on the high virtual channels than messages which have short paths. Because of this, the contention a message faces when competing for buffers differs depending on how far the message must travel in the network.

The same phenomenon exists for the "Crossing node zero" scheme, although the criterion for distinguishing between messages is slightly different. In this case, consider two messages which have the same destination and are at a node common to both of their paths from their sources to their common destination. Again, locally the messages are indistinguishable, as they should, from this point on, follow the same path to their destinations. However, if one of the messages has crossed the zero node previously and the other has not, the two messages will use different virtual channels and, again, face different levels of congestion. Once again, the congestion seen by a particular message at a particular node in the network varies depending on how far apart the message's source and destination are.

An especially disturbing result of this phenomenon is seen with the "Dest < Current" scheme. Two messages being injected into the network from the same node in the same direction, but with different destinations may start out on different virtual channels and face different amounts of congestion during injection. Because of this, it may be easier at some nodes to inject messages which have either short distances to their destinations of long distances to their destinations, depending on the node's location. When some nodes have an easier time injecting their messages into the network than other, starvation becomes a problem. Therefore, with this scheme, a message's likelihood to be starved varies not only with its location in the network, but with the distances that messages it injects into the network need to travel. Fortunately, this problem does not occur in the "Crossing node zero" scheme because all messages at any given node are injected into the same virtual channel.

4 Conclusions

Although the [DS87] scheme provides a simple and efficient method of preventing deadlock in torus networks, it unfortunately introduces non-uniformities in otherwise perfectly uniform networks. The effect of these non-uniformities is quite severe, ranging from differing observed network performance from node to node to a general slowdown of the entire network. Since buffering is critical to achieve good performance in any network, it is not possible to eliminate or minimize the buffering, and thus the problem.

One solution is to associate the buffer space in a node with either the entire node (a shared buffer pool) or with each *physical* channel. Thus, all messages would contend for the same resources, no matter what virtual channel they were routed on. However, in either case, implementation becomes considerably more complex because of the need to create two or more logically separate buffers within one physical buffer.

Because this increases the complexity of the router significantly, it becomes worthwhile to consider abandoning oblivious routing completely and rely on an adaptive technique to prevent deadlock. Non-minimal adaptive routers such as those presented in [NS89, BS92, KS90] prevent deadlock while preserving the node-transitivity of torus networks by relying on local rather than global schemes for preventing deadlock. At the same time, adaptive routing networks allow more flexibility in path selection, making these networks better performers in the presence of uneven network utilization.

References

[BS92] Kevin Bolding and Lawrence Snyder. Mesh and torus chaotic routing. In *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 333–347, March 1992.

[DS86] W. Dally and C. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3), 1986.

[DS87] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[KS90] Smaragda Konstantinidou and Lawrence Snyder. The chaos router: A practical application of randomization in network routing. In *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures*, pages 21–30. ACM, 1990.

[NS89] J. Y. Ngai and C. L. Seitz. A framework for adaptive routing in multicomputer networks. In *Proceedings of the Symposium of Parallel Algorithms and Architectures*, pages 1–9. ACM, 1989.